NAVAL POSTGRADUATE SCHOOL
Monterey, California

(2)

THESIS

A MOBILE ROBOT SONAR SYSTEM

by

Solomon Rand Sherfey III

September 1991

Thesis Advisor: Yutaka Kanayama

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
A MOBILE ROBOT SONAR SYSTEM (U)

12. PERSONAL AUTHOR(S)
SHERFEY, SOLOMON RAND III

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 9/89 TO 9/91	SEPTEMBER 1990	108

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Robotics, sensors

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

In order to function autonomously in the real world a mobile robot must first be able to sense the boundaries of it's operating space. Once the enclosing features and/or obstacles have been sensed they must be interpreted and represented in some way meaningful to the robot's controlling algorithms. The objective of this work is the development of a system of ultrasonic sensors, or sonars, for the mobile robot YAMABICO-11 at the Naval Postgraduate School, and the implementation of a user friendly set of sonar language functions for the robot's control language MML. The sonar hardware includes twelve transducer pairs, their drivers and a bus mounted control card. The sonar control system operates autonomously under direction of the robot's central processor.Extraction of linear features is accomplished by the use of a least-square-fit algorithm of cartesian coordinate pairs to a parametric representation of the including line segment.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Yutaka Kanayama	22b. TELEPHONE (Include Area Code) (408) 646-2095; 22c. OFFICE SYMBOL CS/Ka

A MOBILE ROBOT SONAR SYSTEM

by

Solomon Rand Sherfey III
Lieutenant, U.S. Navy
B.S., Chapman College, 1982

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September, 1991

Author: [illegible]
Solomon Rand Sherfey III

Approved By: [illegible]
Yutaka Kanayama, Thesis Advisor

[illegible]
Mantak Shing, Second Reader

[illegible]
Robert B. McGhee, Chairman,
Department of Computer Science

ABSTRACT

In order to function autonomously in the real world a mobile robot must first be able to sense the boundaries of it's operating space. Once the enclosing features and/or obstacles have been sensed they must be interpreted and represented in some way meaningful to the robot's controlling algorithms. The objective of this work is the development of a system of ultrasonic sensors, or sonars, for the mobile robot YAMABICO-11 at the Naval Postgraduate School, and the implementation of a user friendly set of sonar language functions for the robot's control language MML. The sonar hardware includes twelve transducer pairs, their drivers and a bus mounted control card. The sonar control system operates autonomously under direction of the robot's central processor.Extraction of linear features is accomplished by the use of a least-square-fit algorithm of cartesian coordinate pairs to a parametric representation of the including line segment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

A. Motivation For Sonar Development

Yamabico-11 is one of two mobile robots available for research at the Naval Postgraduate School, the other being the autonomous underwater vehicle of the AUV research group[13]. Yamabico is perhaps better suited for use in basic robotic coursework for three reasons. First, it rolls in the corridor of Spanagel Hall at NPS and doesn't require a team of people and a swimming pool to run a mission. Second, it operates holonomically in two dimensions, greatly reducing the complexity of it's motion and the related control problem. This allows the student to concentrate more on the basics of sensor employment and path planning. Third, a high level language (Mobile robot Motion control Language, or MML) already exists for the student to use for programming the robot's *motion* [9,10].

A set of ultrasonic sonar transducers already existed on Yamabico, and had indeed already been used in research regarding precision navigation [7,8]. A set of functions had been written in C code to utilize the sonars and record data from them. However, when we attempted to use the existing sonar system, we found that it was both hard to understand and extremely fragile. Indeed, we never managed any success with the existing system, failing on several attempts to acquire any usable data. At this point, the decision was made to pursue the design and implementation of a *reliable* and *easily used* sonar system for Yamabico.

B. Functional Goals of the Sonar System

Sensors can be used in two cognitively different ways. First, they provide information about the surrounding environment about which the perceiving entity was previously unaware. In the robotic world we are considering, this might translate to the detection of previously unmapped obstacles in the robot's path. These sensor detections are unplanned events whose occurrence is outside the control of the robot. Their detection, however, can be strongly influenced by the scanning routines employed by the robot and by inference methods employed to determine the probability of such events.

On the other hand, sensors can be employed to verify conditions in the surrounding environment about which the robot is already aware. For example, if the robot is directed to traverse a corridor it is aware, either explicitly or implicitly, that there are two walls parallel to it's intended direction of travel. The explicit knowledge may take the form of a map of the building held by the robot, while the implicit knowledge is inherent in the concept of "corridor". In either event, the robot knows that it may measure the environment and compare the result of those measurements against it's internal representation of the environment in order to assess it's situation.

Our goal is to develop a sonar system which provides the user with the functionality to explore and use the environment in the manner described above. It would be an easy matter to simply say "look everywhere all the time and record everything you see" in order to provide a complete and continuous catalog of Yamabico's environment. The computational resources necessary to accomplish such a task, however, far exceed the capabilities of our machine. We must be much more definitive about *where* we look, *when* we look and *what* we do with the data once we've sensed it. From another perspective, the sensory are a basic task for the robot, just as are the locomotion functions for physically moving itself about. We wish to keep these basic functions on the same level of complexity, providing a homogenous environment for the eventual user of the robot control language. For this reason, complex actions such as an automatic safety sweep of the surrounding area are left to higher level implementations which will use the more basic functions provided here.

C. Design Goals

Our basic design goals may be described as follows:

1) Provide basic sonar data (range and position) with the minimum delay possible.
2) Provide a method of *linear feature extraction* for the description of the robot's environment (presumed, for our purposes, to be orthogonal).
3) Minimize the use of CPU time as much as possible.
4) Maximize the autonomy of the sonar system, thus distributing the processing to some degree.

6) Reduce the complexity of the hardware system in order to improve it's reliability and speed.
5) Provide a user friendly interface in keeping with existing MML functions.

D. Thesis Organization

Subsequent chapters of this work will address our hardware and software design in response to the design goals stated above. Chapter II is devoted to a review of other work related to the implementation of sonars aboard mobile robots.

Chapter III presents the development of the hardware for our project. It begins with a review of what existed originally and traces it's evolution into the new architecture. Chapter IV discusses the extraction of linear features from the environment by means of a least squares fit algorithm, and goes on to describe the means by which data points are selected and filtered for application to the algorithm. Chapter V presents the high level functionality of the sonar control language, the user interface. Also discussed here are the data structures the user needs to be aware of in order to properly use the functions. The background functions which implement the user interface are discussed in Chapter VI along with data structures which are normally hidden from the user. Chapter VII presents the results of testing of the language, including some actual missions run in the corridor and the data returned. Conclusions and avenues for future work are presented in Chapter VIII.

E. Acknowledgment

We acknowledge the invaluable contribution of Mike Williams of the NPS Computer Science Department staff to the design and construction of the sonar system hardware and firmware currently employed aboard Yamabico.

II. RELATED WORK

The bulk of current research in mobile robot sensors is directed towards the implementation of machine vision and tactile sensors. Ultrasonic rangefinding, however, is attractive for it's low cost and relatively simple implementation, making it an excellent research tool for educational institutes. It is also invaluable in environments where optical sensors are occluded (for example, aboard submersible robots).

In a paper published in 1985, Crowley describes a sonar based modeling and navigating system for the IMP mobile robot at Carnegie-Mellon [3]. The proposed sonar system utilizes one transducer with a beam spread of approximately 5 degrees. The transducer is to be rotated in steps of 3 degrees, completing a full revolution in about 10 seconds. The range of the sensor is reported as 25.6 feet with a resolution of 0.10 feet. Crowley develops his sensor model by first converting his range data into cartesian coordinates and then searching for "break" points. Breaks are defined as points where the distance between adjacent coordinate pairs differed by more than a preset constant. The resulting sets of points are fitted to line segments using a recursive routine which compares a constant to the perpendicular distance from individual points to a line drawn between the set's endpoints. The resulting line segments are then linked together to form the world model. Drumheller also conducted research with a single sonar transducer [4]. In his work a Polaroid transducer is mounted at an altitude of 5.5 feet and is rotated in 3.6 degree increments, thus a complete revolution is made in 100 steps. The greater altitude of the sensor over the IMP (which held the sensor at 31 inches above the floor) served to give Drumheller's machine a view of the *room* vice a view of the *furnishings*. Drumheller then extracted line segments from the data in using an iterative endpoint fit. With the segmented representation of the room Drumheller performed pattern matching to determine the sonar's (and thus an attached robot's) location and orientation in the room.

Elfes has conducted research in sonar based mapping and navigation at Carnegie-Mellon [5]. His work is based on the *Neptune* mobile robot, a three wheel device with a circular array of 24 Polaroid ultrasonic transducers. The transducers are separated at an angle of 15 degrees and are at a height of 31 inches above the ground. The Polaroid transducers operate at a frequency of approximately 55 KHz and have a beamwidth across the main lobe of roughly 30 degrees. The machine is *not* autonomous, having a Z80 microprocessor aboard which simply manages the firing of the transducers. The collected data is transferred via a serial link to a VAX mainframe where the interpretation of the data takes place. The focus of Elfes' research is the development of an occupation probability map for an area based on the accumulation of unique views of individual regions. As more views of a region (either from different sonars or from the same sonar at different positions) return an echo from that region the probability of that region being occupied goes up. Also, the resolution of the occupancy map improves as views from different points are collected. Elfes relates some basic problems with the use of ultrasonic sensors, particularly:

- sensitivity sharply declines when the axis of the sonar beam departs from the normal of the reflecting surface
- sonar beams suffer from specularity, or the reflection of the beam between multiple surfaces, causing false range readings
- the relatively wide beamwidth of the sonar beam imposes only a loose constraint on the position of the detected object

Multiple transducer sonar on an autonomous mobile robot is one of the achievements of the HERMIES-IIB robot assembled at the Oak Ridge National Laboratory[1]. HERMIES-IIB mounts 25 Polaroid transducers. 24 of these are mounted as six 2x2 arrays and the remaining sensor is mounted singly as a collision avoidance system. Five of the six arrays are mounted in a rotatable, semi-circular ring on top of the robot; the sixth is mounted on a tiltable platform attached to the rotating ring. The advantage of the 2x2 array organization of the transducers is the reduction of the sonar beamwidth by virtue of phased array operation. Reduced beam width means, of course, enhanced resolution of object location. HERMIES-IIB is a major improvement in mobile computing power over earlier autonomous robots. It boasts an IBM AT microcomputer with both hard and floppy disk drives and 2 Mbytes of RAM. The IBM is host to an eight node NCUBE parallel processing system which executes the navigation and image processing programs (the robot also carries two cameras, in addition to the sonars). The IBM computer communicates to a VME rack by way of an eight megabaud parallel link. The VME rack is loaded with dedicated processors for the operation of the sonar subsystem, motion control, manipulators and other ancillary functions. The application of the sensor systems is interesting in that the sonar system is used for the purpose of navigation, and the vision system for interpreting and operating various mechanical systems and control panels.

Crowley expands his earlier work into systems with multiple sensors [3]. He makes strides in improving the time response of the sonar system to the user by his method of maintaining a "sonar horizon". He utilizes a segment finding method similar to that used in his earlier work, with the advantage that multiple points can be found simultaneously with multiple sensors. The line segments composing the local model are expressed in parametric terms, facilitating later matching with a world map.

The theoretical works by Kuc [11,12] are of interest in that they present the physical basis for the operation of the ultrasonic sensors commonly used aboard mobile robots. Of particular interest is his development of a simulation model, for in that development lies some understanding of peculiar range data distribution at corners and edges that had been observed in previous work on Yamabico.

Early work on our platform, Yamabico, was conducted by Hartman, Kanayama and Smith [7]. In this work, the *least squares fit* method for segment finding is explored and the sonar system used to facilitate precise navigation. The least squares fit algorithm is described in greater detail in a later work by Kanayama and Noguchi [8] and it's application to the NPS AUV project is detailed by Floyd, Kanayama and Magrino in [6].

III. HARDWARE DEVELOPMENT

As pointed out in the introductory chapter, the existing sonar of Yamabico had failed. A functional, reliable sonar system is essential to the continuing research which used Yamabico as a test platform. Evaluation of the current system, our sensor needs and our fabrication abilities led us to the conclusion that a newly designed sonar to replace the existing system was the correct approach. Our design parameters included:

- direct bus interface for data transfer
- reduction of number of circuit cards to one (other than drivers)
- fast enough to reduce positional uncertainty to less than 1 cm.
- use of existing transducers and drivers
- ability to choose between polled and interrupt operation

In this chapter we will first briefly describe the existing sonar system of Yamabico. We will then go on to describe the new control system hardware and it's method of operation.

A. Existing Sonar System of Yamabico-11

Yamabico employs twelve ultrasonic sensors, or sonars, operating at 40 kilohertz and distributed around the periphery of the robot as shown in Figure 3.1 below.

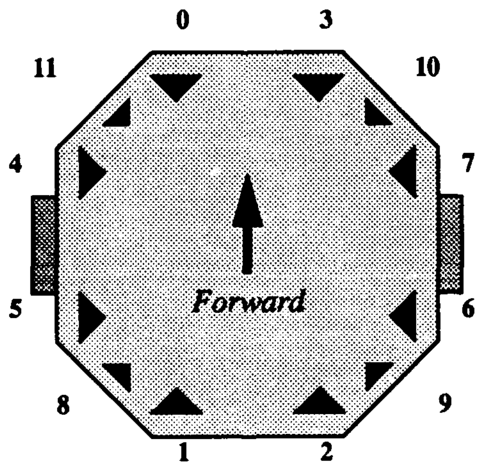

Figure 3.1. Sensor Location

Each sensor is actually a pair of transducers, one to transmit the ultrasonic pulse and another to receive the echo. These sonar transducers are connected to three transmit/receive boards which control four sonars each. These boards amplify the oscillator signal provided by the 6809 processor and apply it to the sonar transmitters, and amplify the received echo to provide an output pulse at TTL levels. The three transmit/receive boards are in turn controlled by a 6809 processor board which times the output signals, gathers the return pulse and forwards the data. The data stream is routed via a ttl/rs232 converter to a serial input / output card, and thence finally to the VME bus and the central processor (see Figure 3.2 below).

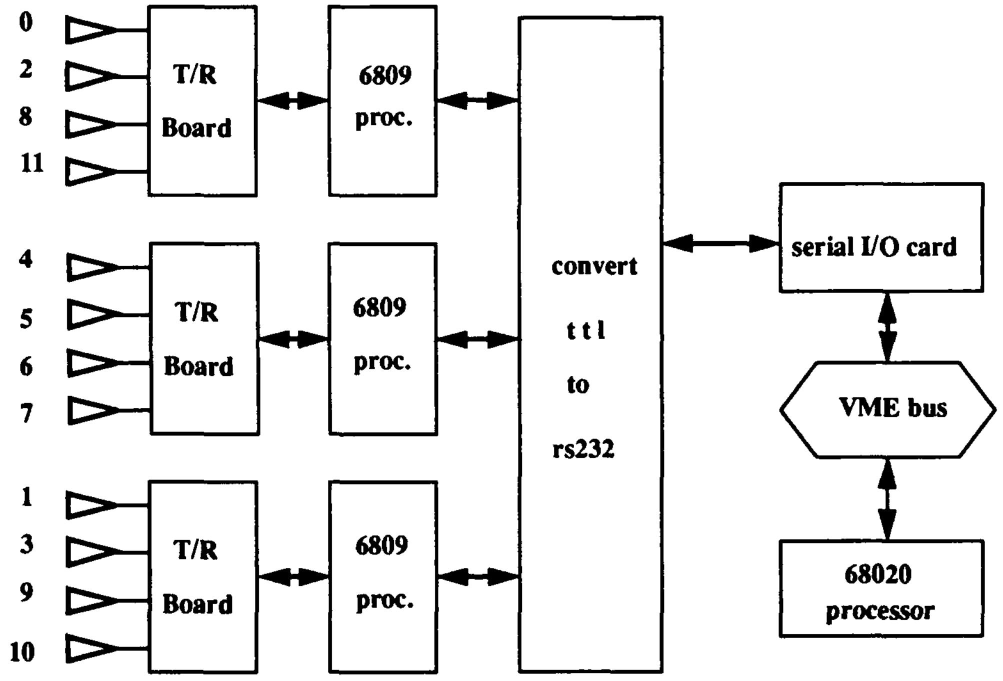

Figure 3.2. Existing Sonar Hardware Architecture

The data for an individual sonar is composed by the 6809 processor and transmitted to the 68020 central processor in a series of four one byte serial transfers, each driven by a separate interrupt. The interrupt handler in the central processor concatenates the single bytes into one long word, which it then breaks up into the actual range data and the sonar number. In addition, the interrupt handler records the robots position in x, y and theta with the transfer of *every* byte, even though only the values recorded with the transfer of the *last* byte are placed on the stack for further use. When four bytes have been collected, the interrupt handler places the data on the stack and calls a C routine to update the sonar table, and then calls another C routine to perform linear fitting and other functions upon the data.

When the system failed to operate, an evaluation was made to determine what the problem might be. As a result of that evaluation, these conclusions were reached:

- the network of circuit cards and cables was far more complex than necessary, contributing to the frailty of the system.
- *if* the system worked, the interrupt driven byte-wise transfer of data would occupy an inordinate amount of processor time, causing system delays.
- existing software was poorly documented and relatively “unfriendly”.

A design group consisting of Yutaka Kanayama, Sol Sherfey and Mike Williams was formed and determined that the construction and implementation of a new sonar system was both called for and within the capabilities of the group and the facilities available to them.

B. New System Design

The design goals were as follows:

- retain the existing transducers and their driver/amplifiers.
- reduce the circuitry between the driver/amplifiers to only one card.
- make sonar data immediately available to the bus, vice following a complex, circuitous (i.e. *slow*) serial path.
- reduce sampling time to a minimum.
- allow for either interrupt driven or polled operation
- improve coding to process data more efficiently and provide the user with a more intuitive programming environment.

The software aspects of the project are detailed in the Software Implementation chapter. In the remainder of this section we will discuss the hardware/firmware aspects of the project. A block diagram of the new sonar system is provided in Figure 3.3.

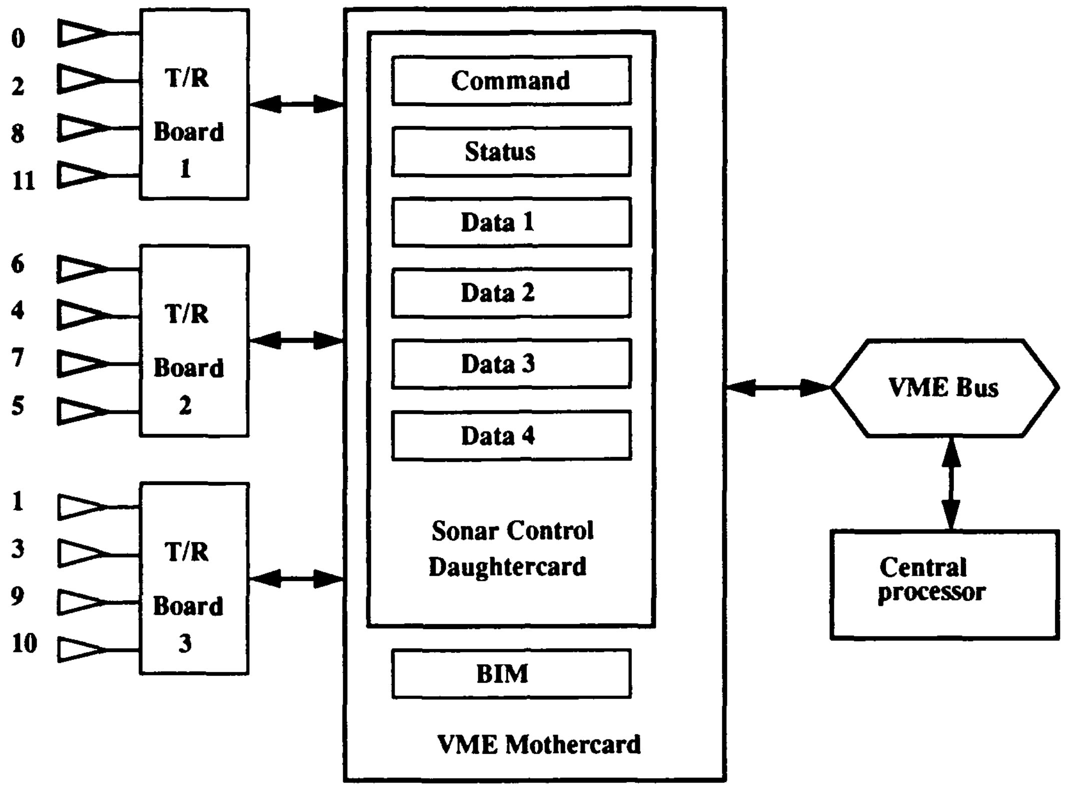

Figure 3.3. New Sonar Hardware Architecture

1. Sonar Grouping

In order to reduce sampling time the sonars are operated in logical groups of four. The sonars of a logical group are all pulsed simultaneously and thus reduce the sampling time by a factor of four as compared to individual firing of the sonars. The sonars of each logical group are oriented in such a way as to:

- prevent mutual interference
- provide a "look" in all four directions from each group
- present a similar aspect from each sonar during a rotational scan

Thus, logical group 0 consists of sonars 0, 2, 5 and 7 (see Figure 1); group 1 of sonars 1, 3, 4 and 6; group 2 of sonars 8, 9, 10 and 11; and group 3 is a "virtual" group which consists of four permanent test values. The axis of each sonar is oriented at 90 degree angles from it's neighbors and the sonars of a group are distributed symmetrically about the robot's axis of rotation.

In addition to being *logically* grouped, the sonars are also *physically* grouped. The physical grouping of the sonars is made to distribute the electrical load over the driver boards evenly and thus minimize any electrical transients associated with operation of the sonar. The physical grouping connects sonars 0, 2, 8 and 11 to driver/amplifier board 1; sonars 4, 5, 6 and 7 to board 2; and sonars 1, 3, 9 and 10 to board 3. The reader will note that pairs of sonars from logical groups are assigned to physical groups, for example, sonars 0 and 2 from logical group 0 are assigned to physical group (driver/amplifier board) 1. To further reduce any power transients associated with sonar operation, the paired sonars are pulsed in opposition to one another, as shown in Figure 3.4.

2. Pulse Control

Initial design of the control circuitry was based on two primary parameters: (1) a desired maximum range of 400 cm. and (2) a pulse width of 1 msec. Assuming a speed of sound in air, at sea level, of 340 meters/second we may calculate a round-trip time:

$$\text{round trip time} = \frac{400 \text{ cm.}}{34000 \text{ cm./sec.}} \times 2 = 23.53 \text{ msec.} \qquad \text{(Eq 3.1)}$$

This round trip time is the period during which a valid echo may be received and is referred to as the *receive gate.* This interval is rounded up to 24 msec. and is derived by division of the sonar system's 2 MHz clock to ensure that the receiver is not falsely triggered by a direct path reception from it's adjacent transmitter, we opt to disable the receiver until the transmit pulse is complete. This will have the disadvantage of setting a minimum range equal to half the distance sound would travel in the time of a transmit pulse.

$$\text{minimum range} = 34000 \text{ cm./sec.} \times 1 \text{ msec.} \times 0.5 = 17 \text{ cm.} \qquad \text{(Eq 3.2)}$$

This minimum range lies approximately 9 cm. outside the periphery of the robot. In order to allow the measurement of objects up to the periphery of the robot, the pulse width was decreased to 0.5 msec thus reducing the minimum range to 8.5 cm.

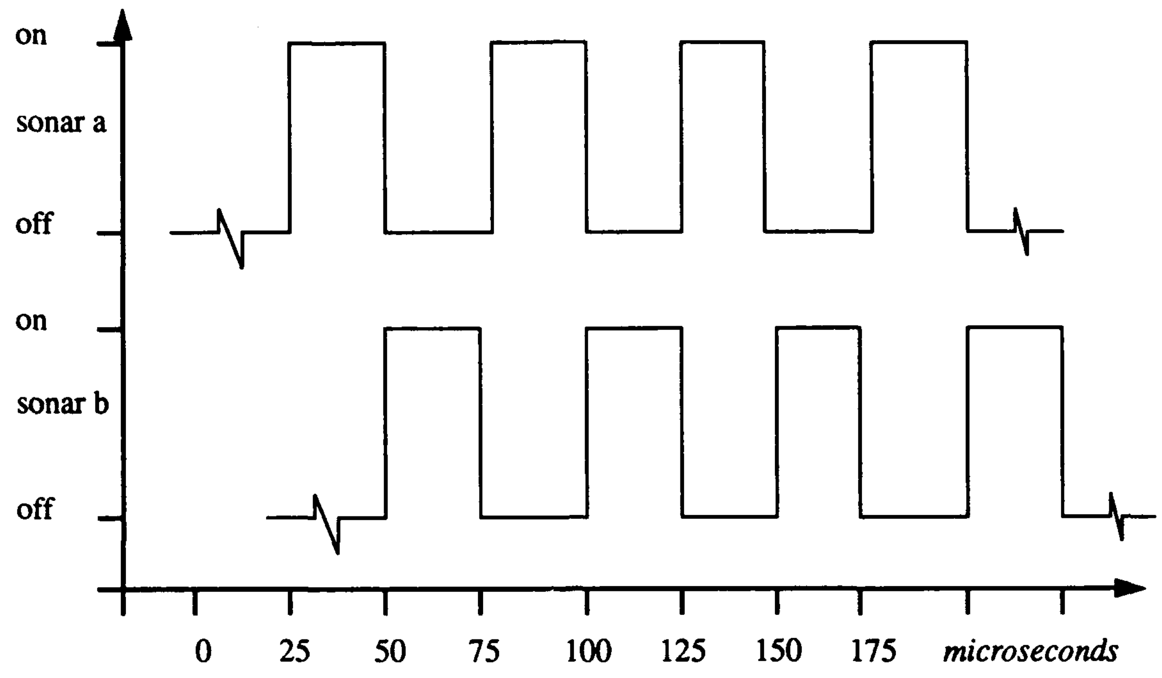

Figure 3.4. Opposed Sensor Firing

In actual practice, the minimum range is set by firmware to 9.6 cm., the additional distance being due to some time being allotted for switching and settling in the circuitry.

All sonars of a logical group are pulsed simultaneously. Which groups are fired is determined by the value of the corresponding bit in the *command register* of the sonar control board, which in turn is set by the user with an MML function. Hence, if bit 2 is set to 1 then group 2 sonars will be pulsed. If more than one group is selected to be pulsed, the sonar control board will pulse the first group on the list, and when the data from that pulse has been read from the fourth data register the sonar control board will proceed to the next group and pulse it, and so on in round robin fashion. Groups with their control bit set to 0 will *not* be pulsed. The sampling rate can thus be as high as 41 Hz with only one group enabled (based on a 24 msec. read gate as determined in equation 3.1) and will be halved for each additional group enabled. At a nominal robot speed of 30 cm/sec this sampling rate could provide an updated range within 0.75 cm. of travel, exceeding our desired positional accuracy of 1 cm. Of course, real performance will be affected by any delay in reading the data registers due to other demands on the central processor (processing the sonar data, controlling motion, etc.).

3. Range Finding

There are four 16 bit data registers on the sonar control board, one for each of the four sonars in a logical group. When the transmit pulse is sent to the driver/amplifier boards a counter is started which increments each of the data registers every 6 microseconds. This time period is equivalent to a range of 1.02 millimeter:

$$\text{range} = 340000 \text{ mm/sec} \times 6 \text{ microsec} \times 0.5 = 1.02 \text{ mm} \qquad \text{(Eq 3.3)}$$

The incrementation of a particular data register continues until an echo is received or the range gate times out. The first 12 bits of the data register are allotted for range accumulation, thus allowing for a maximum range of 4.177 meters (4095 x 1.02 mm). If the range gate should time out before an echo is received, the high bit of the over ranged sonar's data register is set to 1. This is the "overrange" bit and is used to signal the ensuing software that no echo was received. Bits 12, 13 and 14 of the data registers are not used. When the ranging cycle is complete, the appropriate group number is written into bits 4 and 5 of the status register and the "ready" bit, bit 7 of the status register, is set to 1. The ready bit is used as a flag when operating in the polled mode; i.e. without interrupts.

4. Interrupt Control

The sonar control board is actually a daughtercard which rides on a VME bus mothercard. The mothercard carries address decoders, bus drivers and interrupt control circuitry in the Bus Interface Module (BIM).

When the sonar has completed a ranging cycle an interrupt request is provided to the BIM. The BIM's *control register* holds information which determines whether an interrupt is to be generated or not, and if so which interrupt level is to be generated. Presuming an interrupt is generated, when the correct acknowledgment returns on the address lines the BIM's *vector register* provides the vector table entry where the central processor may find the vector to the interrupt handler. The correct interrupt level, the interrupt enable bit and interrupt vector are loaded to the BIM during software initialization.

5. Data Transfer

Each of the data registers is individually addressed on the VME bus by a VME short address, as is the status register. Transferral of the data is extremely straightforward. The interrupt handler simply reads the correct register, masks out the unwanted bits and writes the data to the stack. When the last data register is read, the sonar system resets the data registers and commences a ranging cycle on the next sonar group in it's round robin. The system will continue to operate autonomously until all the sonars are disabled.

IV. LINEAR FEATURE EXTRACTION

In addition to simple *range* and *point position* data, we desire the sonar system to develop representations of *linear features* in an orthogonal world. To do so we must provide some method for recognizing sets of data points which form the linear feature and a method for finding and describing the line segment that best fits that set of data points. This is accomplished in reverse fashion, i.e. we presume the data we are receiving belongs to such a set and continuously modify a descriptive line segment to a best fit of the data using a least squares fitting algorithm. This line segment continues to grow until the incoming data or certain measures of the line segment indicate that the line segment should be ended and a new one started. We use an implementation of least squares fitting described by Kanayama and Noguchi [8].

A. Least Squares Fitting

Suppose we have collected n consecutive valid data points in a local coordinate system, $(p_1,\dots, p_n)$, where $p_i = (x_i, y_i)$ for $i = 1,\dots,n$. We obtain the moments m_{jk} of the set of points

$$m_{jk} = \sum_{i=1}^{n} x^j_i y^k_i \qquad (0 \le j, k \le 2, \text{ and } j + k \le 2) \qquad \text{(Eq 4.1)}$$

Notice that $m_{00} = n$. The centroid C is given by

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}}\right) \equiv (\mu_x, \mu_y) \qquad \text{(Eq 4.2)}$$

The secondary moments around the centroid are given by

$$M_{20} \equiv \sum_{i=1}^{n} (x_i - \mu_x)^2 = m_{20} - \left(\frac{m_{10}}{m_{00}}\right)^2 \qquad \text{(Eq 4.3)}$$

$$M_{11} \equiv \sum_{i=1}^{n} (x_i - \mu_x)(y_i - \mu_y) = m_{11} - \left(\frac{m_{10} m_{01}}{m_{00}}\right) \qquad \text{(Eq 4.4)}$$

$$M_{02} \equiv \sum_{i=1}^{n} (y_i - \mu_y)^2 = m_{02} - \left(\frac{m_{01}}{m_{00}}\right)^2 \qquad \text{(Eq 4.5)}$$

We adopt the parametric representation (r, α) of a line with constants r and α. If a point p = (x,y) satisfies an equation

$$r = x\cos\alpha + y\sin\alpha \qquad (-\pi/2 < \alpha \leq \pi/2) \qquad \text{(Eq 4.6)}$$

then the point p is on a line L whose normal has an orientation α and whose distance from the origin is r (Figure 4.1). This method has an advantage in expressing lines that are perpendicular to the X axis. The point-slope method, where y = mx + b, is incapable of representing such a case (m = ∞, b is undefined).

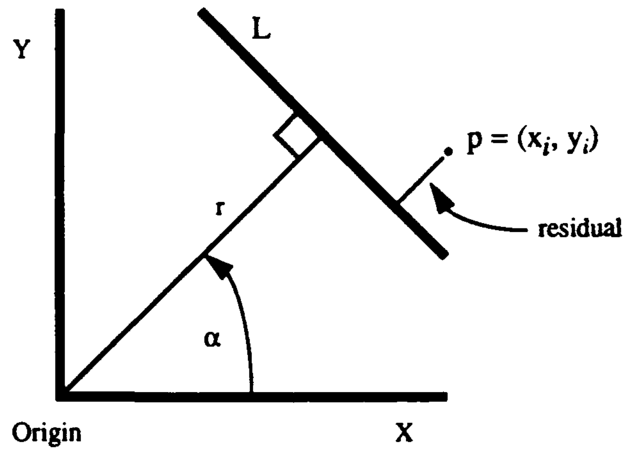

Figure 4.1. Representation of a line *L* using *r* and α.

The residual of point p_i = (x_i, y_i) and the line L = (r,α) is $x_i\cos\alpha + y_i\sin\alpha - r$. Therefore, the sum of the squares of all residuals is

$$S = \sum_{i=1}^{n} (r - x_i\cos\alpha - y_i\sin\alpha)^2 \qquad \text{(Eq 4.7)}$$

The line which best fits the set of points is supposed to minimize S. Thus the optimum line (r,α) must satisfy

$$\frac{dS}{dr} = \frac{dS}{d\alpha} = 0 \qquad \text{(Eq 4.8)}$$

Thus,

$$\frac{dS}{dr} = 2\sum_{i=1}^{n} (r - x_i \cos\alpha - y_i \sin\alpha) \qquad \text{(Eq 4.9)}$$

$$= 2\left(r\sum_{i=1}^{n} 1 - \left(\sum_{i=1}^{n} x_i\right)\cos\alpha - \left(\sum_{i=1}^{n} y_i\right)\sin\alpha\right)$$

$$= 2\,(rm_{00} - m_{10}\cos\alpha - m_{01}\sin\alpha)$$

$$= 0$$

and

$$r = \frac{m_{10}}{m_{00}}\cos\alpha + \frac{m_{01}}{m_{00}}\sin\alpha = \mu_x \cos\alpha + \mu_y \sin\alpha \qquad \text{(Eq 4.10)}$$

where r may be negative. Substituting r in Equation (4.7) by Equation (4.10),

$$S = \sum_{i=1}^{n} \left((x_i - \mu_x)\cos\alpha + (y_i - \mu_y)\sin\alpha\right)^2 \qquad \text{(Eq 4.11)}$$

Finally,

$$\frac{dS}{d\alpha} = 2\sum_{i=1}^{n} \left((x_i - \mu_x)\cos\alpha + (y_i - \mu_y)\sin\alpha\right)\left(-(x_i - \mu_x)\sin\alpha + (y_i - \mu_y)\cos\alpha\right)$$

$$= 2\sum_{i=1}^{n} \left((y_i - \mu_y)^2 - (x_i - \mu_x)^2\right)\sin\alpha\cos\alpha + 2\sum_{i=1}^{n} (x_i - \mu_x)(y_i - \mu_y)(\cos^2\alpha - \sin^2\alpha)$$

$$= (M_{02} - M_{20})\sin 2\alpha + 2M_{11}\cos 2\alpha \qquad \text{(Eq 4.12)}$$

$$= 0$$

Therefore

$$\alpha = \frac{\operatorname{atan}\left(2M_{11} / (M_{02} - M_{20})\right)}{2} \qquad \text{(Eq 4.13)}$$

Equations (4.10) and (4.13) are the solutions for the line parameters generated by a least squares fit.

B. Thinness Testing

The equivalent ellipse of inertia for the original n points is an ellipse which has the same moments around the center of gravity. M_{major} and M_{minor} are moments about t':e major and minor axes respectively (Figure 4.2).

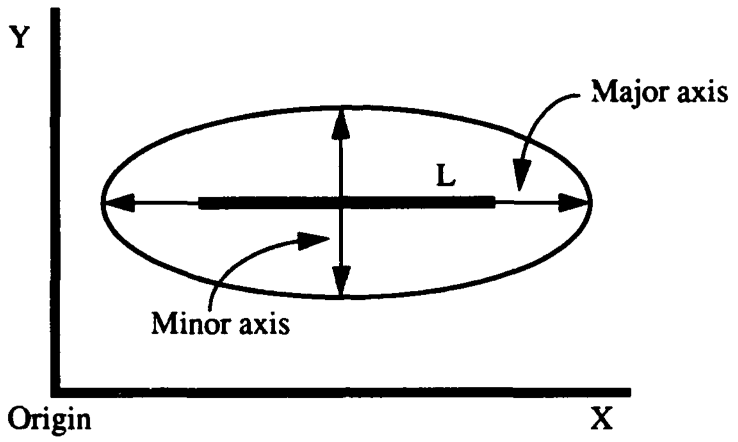

Figure 4.2. The *equivalent ellipse of inertia* for line *L*.

$$M_{major} = (M_{20} + M_{02})/2 - \sqrt{(M_{02} - M_{20})^2/4 + M_{11}^2} \qquad \text{(Eq 4.14)}$$

$$M_{minor} = (M_{20} + M_{02})/2 + \sqrt{(M_{02} - M_{20})^2/4 + M_{11}^2} \qquad \text{(Eq 4.15)}$$

The diameters d_{major} on the major axis and d_{minor} on the minor axis of the equivalent ellipse are

$$d_{major} = 4\sqrt{M_{minor}/m_{00}} \qquad \text{(Eq 4.16)}$$

$$d_{minor} = 4\sqrt{M_{major}/m_{00}} \qquad \text{(Eq 4.17)}$$

We define ρ, the ellipse thinness ratio, to be the ratio of d_{minor} and d_{major}:

$$\rho = \frac{d_{minor}}{d_{major}} \qquad \text{(Eq 4.18)}$$

A small ρ means a thin ellipse; as ρ increases toward 1 the ellipse degrades to a circle representing a thick line or a "blob" of points. We will use ρ as an additional measure of the linearity of a set of points and, by comparing ρ to a constant C3, we may use ρ to determine the end of a line segment.

C. Finding Endpoints

The residual of a point $p_i = (x_i, y_i)$ is

$$\delta_i = (\mu_x - x_i)\cos\alpha + (\mu_y - y_i)\sin\alpha \qquad \text{(Eq 4.19)}$$

Therefore, the projection, p'_i of the point p_i onto the major axis is

$$p'_i = (x_i + \delta_i \cos\alpha, y_i + \delta_i \sin\alpha) \qquad \text{(Eq 4.20)}$$

We will use p'_i and p'_n as estimates of the endpoints of the line segment L obtained from the set p of data points.

D. Residual Testing

In addition to the *ellipse thinness* testing which occurs after a new point has been included in the line segment, we wish to do some pre-filtering of the data in order to remove points from the data stream which are clearly *not* colinear with the existing points of set p. In this way we can often detect the end of a line segment before having to perform the considerable computations necessary to include it in the line. If the point satisfies

$$\delta_{i+1} < \max(\sigma \times C1, C2) \qquad \text{(Eq 4.21)}$$

where C1 and C2 are positive constants (typically, C1 = 2.0 and C2 = 2.0) and the standard deviation σ is

$$\sigma = \sqrt{M_{minor} / (i-2)} \qquad \text{(Eq 4.22)}$$

then the point can be included in the current line segment.

E. Beginning Line Segments

Clearly, at least two data points must be collected in order to define the start of a line segment. In the software model adopted in this project, the usefulness of data points to the current line segment is judged on a "best two out of three" basis. In this model, two out of three consecutive data points must fail the residual testing in section D above in order for the system to end the current line segment. In keeping with this model, we have chosen three as the number of points necessary to start a line segment.

With the line segment established, collection and testing of the fourth data point can proceed. If the data point *passes* the residual testing, the moments and test values for the line are calculated including the new point and the ellipse thinness test performed. Should that test pass, the line segment parameters (endpoints, length, etc.) are updated and the system proceeds to gather a new data point.

If, however, the fourth data point should *fail* the residual testing, the system deletes the first data point gathered and restarts the line segment with the second, third and fourth data points. This process continues until the first data point collected following line segment initiation passes residual testing. In this way we eliminate *erratic* start-up data, but start the line segment with the *earliest possible* acceptable data.

F. Ending Line Segments

There are three ways in which a line segment is ended. It may be ended by the failure of data points to pass the residual testing, by the failure of the line segment to pass ellipse thinness testing, or explicitly ended by the user of the program.

In the case of ellipse thinness testing, ρ is compared to a constant C3. If ρ is greater than C3, the line segment is ended.

In the case of residual testing, we wish to protect ourselves from the effects of infrequent erroneous data points. If we were to end a line segment on the strength of one data point that failed residual testing, a noisy environment would quickly reduce linear features to an unmanageably large number of segments. As a protection against this occurrence, we require that two out of three consecutive data points fail to pass residual testing before ending a line segment.

If a data point fails residual testing, the system does nothing but store the data point temporarily and then gathers two more data points. If either one of the second pair data points fails residual testing, the line segment is ended and the failed data points form the start of a new line segment. If both of the second pair of data points pass residual testing, the errant data point is thrown out and the two succeeding data points are sequentially fed to the linear fitting algorithm.The process then carries on normally.

V. SONAR USER INTERFACE

With the physical design of the sonar system settled and the basic capabilities of the system defined, we turn our attention to the implementation of the sonars functions in software. In this chapter we will first examine our basic precepts for the design of such a software system, and follow that with synopsis of the functions written to support our goals.

A. Precepts

The user interface must clearly represent the high level tasks the user wishes to accomplish and transmit the correct instructions to the mid level code to perform those tasks. Those high level tasks include:

1) providing the range to an object on demand
2) providing global x,y coordinates for sonar returns on demand
3) developing a representation of surfaces it may encounter
4) providing those surface representations on demand
5) recording desired data for download and analysis

The sonar system must accomplish these tasks in a real time environment using it's single onboard processor, which must also handle all locomotion processing *and* any higher level functions the user may ultimately develop. Clearly our design must minimize the processing required whenever possible - it is not feasible to simply perform *all* the functions *all* the time. Our language must, therefore, provide a method for enabling functions as needed and disabling those functions when the user no longer requires them.

We must also design a language that is similar enough to the existing MML to form a "seamless" programming environment for the user. Not only does this mean that the sonar functions should be similar in format to the MML locomotion functions, but they should be similar in *scope*. The user should feel that he is accomplishing the same level of control with the sonar functions as he is with the locomotion functions. As an example, it would be appropriate to issue a sonar command to gather data from a given sonar following a locomotion command to move from one point to another. It would be inappropriate to issue a sonar command to "map the enclosing space" following a list of move commands to traverse the space. A command with such wide scope is better suited to higher level cognitive functions, such as a *navigator*, rather than the sensory level.

Finally, the user interface should reveal only the level of detail necessary for the user to effectively employ the sonars. Underlying processes and data structures should remain hidden as much as possible, leaving function calls that are as simple and easy to understand as possible.

B. Sonar User Language Overview

1. Range and Position

In order to reduce the time needed to obtain a range value and to avoid firing all the sonars, an *enable_sonar* function is provided. After a sonar group is selected by this function, a background process repeatedly fires the sonar group at a constant interval and maintains the range values for the four sonars in the group in a data structure called *sonar_table*. These current range values may be individually requested by a *sonar* function call, or the robot user may call the *wait_sonar* function to delay further processing until a new range update is available from the sonar. A *disable_sonar* function allows the user to turn off a sonar group when it is no longer needed, thus allowing more frequent sampling of the remaining enabled sonars.

While a simple range is useful for tasks such as obstacle avoidance and wall following, developing a sense of position in the real world requires that we fix the position of those sonar returns in some sort of coordinate system. To accomplish this we provide a *global* function call, which returns a data structure called *posit* containing the global x and y coordinates of the origin of the latest sonar echo, and the orientation of the sonar axis with respect to the global x axis at the time of the range.

2. Linear Features

If the user desires to acquire a surface rather than simply a range, the *enable_linear_fitting* function may be called. When invoked for a particular sonar, this function will cause the best straight line fit for a continuing sequence of range values from that sonar to be found. When the range data falls outside of preset bounds, the linear feature being generated will be terminated and a new segment begun. The completed linear feature is described in a data structure called a *segment* which, in turn, is stored in a *segment_list* for that sonar. A *disable_linear_fitting* function allows the user to discontinue linear fitting when not needed. A *set_parameters* function allows the user to specify certain parameters for the least squares line fitting routine used to generate the linear features. A *finish_segments* function allows the user to complete segments at the end of a particular motion of the robot, when no other indication exists to cause the ending of the segment.

As noted above, completed *segments* are stored in a list of such structures for each sonar. The user may access these structures with a *get_segment* function call, which returns the *segment* at the head of the list and moves the head pointer to the next descriptor. The descriptor returned is the oldest descriptor on the list; successive calls to *get_segment* produce successively more recent descriptors until the most recent is sent and the head pointer goes to null. If the user needs the data for the linear feature currently being assembled by the background process, a function called *get_current_segment* is provided. The need for such a function arises from the need for current data for navigational updates.

3. Data Logging

Since Yamabico is, after all, a research vehicle it will be necessary for the robot to communicate what it measures with it's sonars back to a host machine and the user. This operation is accomplished in two steps. First, the data to be logged is selected and the data logging enabled by the *enable_data_logging* function. The data is stored in arrays according to the type of data selected for logging. These arrays are the *raw_data_log,* the *global_data_log* and the *segment_data_log.* There are four of each type of data array, for a total of twelve data files. At the end of the robot's mission the collected data may be transferred to the host by use of the *xfer_raw_to_host, xfer_global_to_host,* and *xfer_segment_to_host* functions, respectively. Since the sonar system can collect data at intervals as short as 25 milliseconds, it may be desirable (or necessary) to record only a portion of the data collected. To accomplish this the *set_log_interval* function is provided. Of course, as with the other enabling functions, data logging can be stopped with the *disable_data_logging* function.

4. Program Control

When the user writes his program for the robot, he may find that he needs to "stall" the progress of the program at times to allow the robot to physically "catch-up" to the real world position that the program presumes it is in. This is necessitated by the method of operation of the locomotion functions. As *sequential* locomotion functions (ex. *move, rotate, and stop*) are encountered in the user program, they are processed onto a queue and the processing of the user program continues. The function at the head of the queue is performed until it's goal has been met, at which time it is popped off the queue and the next sequential function is commenced. The result is that although the user writes a sonar function after, say, the third move command in his program, in actuality it may occur during the execution of the first command. To counter this effect, the designers of the locomotion functions included mark_motion and wait_motion commands to halt further processing of the user program until a specific sequential command was complete. In our sonar language, we incorporate a similar feature to halt processing of the user program until certain conditions are met. This is the *wait_until* command, which can delay processing based on the robot's x, y, or theta or based on the range from a given sonar. Additional functions available to the user to assist in control of the robot are the *enable_* and *disable_interrupt_operation* functions. With these functions the user can shift from interrupt driven operation to polling operation and back, thus allowing the user greater latitude in shaping the operation of the robot.

C. Data Structures Synopsis

All of the data structures used in *sonar.c* are of a fixed size and are established at compile time. Since the functions always deal with the same data structures, differentiating only by sonar number within the structures, those structures were made global in scope and parameter passing is often limited to a single integer representing the sonar number. All of the data structures are defined in *mml.h* in such a way that the structures are actually declared in *main.c* and referred to as external in other files, including *sonar.c*. In the remainder of this section we will present a brief description of all the data structures specific to *sonar.c*. and of interest to the user.

1. Sonar Table

Structure:

```
typedef struct{
 int      file[3],
          fitting,
          global,
          update,
          interval,
          filenumber[3];
double    d,
          x,
          y,
          t,
          gx,
          gy,
          offset,
          phi,
          axis;
          } SONARD;
SONARD sonar_table{16};
```

Description: As can be seen from Figure 6.1, the sonar table is central to the operation of the sonar system. It contains not only the range (d) but the robot's position at the time of the range (x, y and t) and the global coordinates corresponding to that range and position (gx and gy, if global conversion is enabled). The sonar table is also the location of constants describing the position of the individual sonar relative to the robot's coordinate system (offset, the euclidean distance from robot center to sonar center; phi, the angular offset from the robot's x-axis to the sonar center; and axis, the angular orientation of the sonar beam's axis to the robot's x-axis). The sonar table also contains a number of flags which guide the operation of the sonar system. These are file[3]

and filenumber[3], which hold information for selecting and logging data; interval, which describes how often to log raw or global data; fitting, which directs the linear fitting of data; global, which directs the global conversion of data; and update, which informs the sonar system that new data exists in d.
An array of sixteen of these structures is formed, which is then indexed by sonar number.

2. Segment Descriptors

Structure:

```
typedef struct{
int         sonar;
double      headx,
            heady,
            tailx,
            taily,
            phi,
            r,
            length,
            dmajor,
            dminor;
            } LINE_SEG;
LINE_SEG seg_list[16][5];
LINE_SEG segstruct;
int seg_list_head[16];
int seg_list_tail[16];
```

Description: The LINE_SEG structure contains all the data necessary to completely describe a line segment. This includes an integer to represent the sonar which recorded the segment, and doubles to record the endpoints (head x and y, tail x and y), the angle and length of a normal to the segment from the origin (phi and r), the length of the line segment and the length of the axes of the ellipse containing all the data points of the segment (dmajor and dminor). These structures are arranged in a two dimensional array. One index is the number of the sonar from which the segment is derived; the other index has two pointers, seg_list_ head and seg_list_tail. The nth position of each of these arrays holds an integer (0 through 4) which points to a position in the nth array of seg_list. By using these pointers a circular queue is formed of each of the arrays of seg_list which can hold the 5 most recent segments described by a given sonar. It is presumed that any navigation program will not require more history than these five segments; if so, the second index of seg_list can be increased. The individual LINE_SEG structure called segstruct is a temporary storage location used by *get_current_segment* and *end_segment*.

3. Data Logs

Structure:

```
typedef struct{
int         count,
            next;
double      darray[MAXRAW],
            xarray[MAXRAW],
            yarray[MAXRAW],
            tarray[MAXRAW];
            } RAW;
RAW raw_data_log[4];

typedef struct{
int         count,
            next;
double      xarray[MAXGLOBAL],
            yarray[MAXGLOBAL];
            } GLOBAL;
GLOBAL global_data_log[4];

typedef struct{
int         count,
            next;
double      LINE_SEG array[MAXSEGMENT];
            } LINES;
LINES segment_data_log[4];
```

Description: The data logs are arrays to which the user program writes data during it's execution. These logs are converted to ASCII strings at the completion of the user program by the *xfer_data_to_host* functions, and those strings are in turn transferred to the host by the *host_xfer* function. There are three types of data logs, the raw_data_log, the global_data_log and the segment_data_log, which are referred to by their respective type numbers 0, 1 and

2 when using the *enable_data_logging* function. For each log type there are four structures, or data files, in the respective array. These are referred to by their respective file numbers 0, 1, 2 and 3 in the *enable_data_logging* function. The size of the arrays within the structures limits the number of data elements that may be logged in each datafile.These array size values MAXRAW, MAXGLOBAL and MAXSEGMENT are defined in mml.h. The count value is the number of data values reported to the logging function, while the next value is the number that have actually been logged (these values will differ if log_interval is set to a value other than 1). The raw_data_log records range and the robot's x, y and t positions at the time of the range. The global_data_log records global x and y values for sonar returns. The segment_data_log records line segments in the form of segment descriptors previously described.

4. Posit

Structure:

```
typedef struct{
double     gx,
           gy,
           psi;
           }posit
```

Description: This structure is used to pass the global coordinates of a sonar return and the orientation of the sonar axis with respect to the global x axis back to the requesting function.

D. Definitions Synopsis

There are some definitions made in the mml.h which may make the programming of some of the functions easier. These mnemonics may be used in place of the integer parameters called for by the function synopsis.

1. Sonar Numbers

FRONTL	0
FRONTR	3
LEFTF	4
LEFTB	5
RIGHTF	7
RIGHTB	6
BACKL	1

BACKR	2
FRONTLEFT	11
BACKLEFT	8
BACKRIGHT	9
FRONTRIGHT	10
TESTOVERFLOW	12
TEST1024	13
TEST2048	14
TESTDELAY	15

2. Wait Until Parameters

X	12
Y	13
A	14
D0	0
D1	1
D2	2
D3	3
D4	4
D5	5
D6	6
D7	7
D8	8
D9	9
D10	10
D11	11
GT	15
LT	16
EQ	17
PI	$\pi = 3.14159265358979323846$
DPI	2π
HPI	0.5π
PI34	0.75π
PI4	0.25π

E. User Function Synopsis

Complete code of the sonar user functions is contained in Appendix A. For each function we provide here a synopsis of the functions syntax and a brief description of what the function accomplishes. Where an integer *n* is specified as a parameter to indicate a sonar number, the user may instead use predefined values such as FRONTL or BACKRIGHT. These definitions are made in mml.h and will be presented in the next section, Data *Structures*. Other such definitions exist and will be noted in the appropriate synopsis.

1. Enable Sonar

Syntax:	void enable_sonar(n) int n;
Description:	Causes sonar *n* to pulse and receive echoes. Range data recorded in *sonar_table*. User should remember that *all* the sonars in the logical group to which sonar *n* belongs will pulse simultaneously.

2. Disable Sonar

Syntax:	void disable_sonar(n) int n;
Description:	Turns off sonar *n*. User should remember that the group will continue to pulse if any other sonars in that group are enabled.

3. Sonar

Syntax:	double sonar(n) int n;
Description:	Returns the latest range value for sonar *n* in centimeters as a floating point number. Returns -1.0 if sonar was over-ranged and 0.0 if range was less than 10 centimeters.

4. Wait Sonar

Syntax:	double wait_sonar(n) int n;
Description:	Causes processing to wait until a new value of range for sonar *n* is received, then returns range as in sonar(n).

5. Global

Syntax:	posit global(n) int n;
Description:	Returns the structure posit, which contains global x and y values for the origin of the last sonar echo and sonar axis

orientation at time of range, for sonar *n*.

6. **Enable Linear Fitting**

Syntax: void enable_linear_fitting(n)
int n;

Description: Causes linear fitting routine to find best straight line through the set of data points collected from sonar. Stores segments in *seg_list*, which can store up to five segments. See the *seg_list* synopsis.

7. **Disable Linear Fitting**

Syntax: void disable_linear_fitting(n)
int n;

Description: Stops the linear fitting of data for sonar *n*.

8. **Enable Data Logging**

Syntax: void enable_data_logging(n,filetype,filenumber)
int n, filetype, filenumber;

Description: Sets the correct file flag for sonar *n* to cause a particular data type to be logged into a file designated by *filenumber*. A *filetype* value of RAW will cause raw sonar data to be logged, GLOBAL will cause global coordinates to be logged, and SEGMENT will cause segments to be logged. There are four files for each filetype, labeled as 0, 1, 2 and 3. For example:

enable_data_logging(RIGHTF,GLOBAL,2);

will cause global coordinates for sonar 7 to be logged in file number 2. The integer values of RAW, GLOBAL and SEGMENT are defined in mml.h.

9. **Disable Data Logging**

Syntax: void disable_data_logging(n,filetype)
int n, filetype;

Description: Stops the logging of data *filetype* for sonar *n*.

10. **Get Segment**

Syntax: LINE_SEG *get_segment(n)
int n;

Description: Returns a pointer of type LINE_SEG to the oldest segment in array seg_list for sonar *n*. Function is destructive; i.e. it will move the head pointer to the next segment in seg_list when it returns the pointer to the oldest segment, thus successive calls to this function will return subsequent segments until seg_list is empty. If get_segment is called on an empty list a null pointer will be returned.

11. Get Current Segment

Syntax:

```
LINE_SEG *get_current_segment(n)
int n;
```

Description: Returns a pointer to the segment currently under construction if there is one, else returns a null pointer. It does this by calling end_segment and returning a pointer to the temporary data structure that end_segment constructs. The line segment is not ended by this function call - it will continue to grow until the linear fitting algorithm determines it should stop. This function merely takes a "snapshot" of the segment as it exists at the moment.

12. Set Parameters

Syntax:

```
void set_parameters(c1,c2,c3)
double c1, c2, c3;
```

Description: Allows the user to adjust constants which control the linear fitting algorithm. C1 is a multiplier for standard deviation and C2 is an absolute value; both are used to determine if an individual data point is usable for the algorithm. C3 is a value for ellipse thinness; it is used to determine the end of a segment. Default values are set in main.c to 3.0, 5.0 and 0.1 respectively.

13. Enable Interrupt Operation

Syntax:

```
void enable_interrupt_operation()
```

Description: Places the sonar control in the interrupt driven mode, which is the default mode.

14. Disable Interrupt Operation

Syntax:

```
void disable_interrupt_operation()
```

Description: Causes the sonar control board to cease generating interrupts. Bit seven in the status register is set when there is data ready, and it is the user's responsibility to poll the system.

15. Set Log Interval

Syntax:

```
void set_log_interval(n,d)
```

int n, d;

Description: Sets the value of interval for sonar *n* to *d*. This causes the data logging function to record one data point for every *d* points sent by sonar n. Effective for raw and global data only, has no effect on the logging of segments. Default value is 13, which for a speed of 30 cm/sec and a sonar sampling rate of 40 Hz (one group enabled) results in a data point every 10 cm.

16. Xfer Raw To Host

Syntax:
```
void xfer_raw_to_host(filenumber,filename)
int filenumber;
char *filename;
```
Description: Causes raw data from a file *filenumber* to be downloaded to the host. *Filename* must be entered in double quotes ("dumpraw" for example).

17. Xfer Global To Host

Syntax:
```
void xfer_global_to_host(filenumber,filename)
int filenumber;
char *filename;
```
Description: Same as xfer_raw_to_host, but for global data vice raw data.

18. Xfer Segment To Host

Syntax:
```
void xfer_segment_to_host(filenumber,filename)
int filenumber;
char *filename;
```
Description: Same as xfer_raw_to_host, but for segment data vice raw data.

19. Finish Segments

Syntax:
```
void finish_segments(n)
int n;
```
Description: Completes segments for sonar *n* at the end of a data run. Necessary because the linear fitting function only terminates a segment based on the data - it has no way of knowing that the user has stopped collecting data.

20. Wait Until

Syntax:
```
void wait_until(variable,relation,value)
int variable, relation;
double value;
```

Description: Function will delay it's completion (and thus the continuance of the program it's embedded in) until the variable achieves the relation with the value specified. For example, presume the robot is traveling along the X axis. If the user wants the robot to begin producing sonar data when the robot's x position exceeds 500 cm., he would insert this command after the move command:

```
wait_until(X,GT,500.0);
enable_sonar(sonar_number);
```

The variables are predefined as X, Y, A, and D0 through D11, and correspond to the robot's x position, y position, theta, and range from sonars 0 through 11. Relations are predefined as GT, LT and EQ corresponding to greater than, less than and equal to. Value may be any number expressed as a double or the predefined values PI, HPI, PI34, PI4 or DPI.

VI. BACKGROUND SOFTWARE IMPLEMENTATION

With both the high level and low level interfaces established, we now turn our attention to the mid level code that forms the bulk of the software system. This code manipulates the data provided by the low level hardware as directed by commands from the high level functions. The results of these manipulations are either made directly available to the user or are logged in memory for later transfer to the host computer. We will first examine the precepts for the mid level design, then present an overview of system operation and a synopsis of the background functions.

A. Precepts

The background software system must perform the work specified by the user through the *high level commands* upon the raw data provided by the hardware (*low level*) system. This work, or *mid level tasks*, includes:

1) *interpreting* the high level commands
2) *controlling* the hardware/software system
3) *gathering* the correct data
4) *processing* the data into the desired forms
5) *returning* data as requested
6) *storing* data when directed

As pointed out in the last chapter, Yamabico has only a single processor with which to perform all of the computation for the robot. It is essential, therefore, that we design our code to be as conservative as possible with processor time. We pursue this goal along two avenues. First, we *selectively* process the data rather than *universally* processing it. By this we mean that we perform only the calculations necessary to provide the data the user specifically asks for, rather than performing calculations over an entire set of data. As an example, rather than routinely calculating global coordinates for all sonar returns, we perform those calculations only for those sonar returns designated for linear fitting, global data logging or return of global coordinates to the user. Second, we must design functions in such a way as to minimize processing under "normal" conditions and degrade to more lengthy procedures as it's operating environment becomes more complex. As an example, the linear fitting algorithm should loop through the minimum code possible while the input data is all continuously acceptable. If the robot's sonar data becomes less reliable, then additional testing of the data should be performed only as long as required to filter errant data points from the input stream. When the data again becomes routinely acceptable, the processing should return to the minimum level possible.

We must also account for the interrupt driven nature of the machine in the design of our software. It would be a simple matter to program a machine which had only to deal with sonar processing for one sonar, but a significantly more complex problem to deal with twelve sonars on a time shared basis with locomotion and I/O processes. Using the linear fitting algorithm as an example once again, we note that we must preserve the value of several summations between calls to the sonar system which provide the data points, and must do so for each sonar we are evaluating. In a single purpose processor this interim storage would not be necessary.

B. Background System Overview

The functions discussed in the previous chapter are the *user* functions - the functions normally available to the user for writing his programs. To support these user functions there are a set of *background functions* which the user would not normally use or even have access to. At the root of these background functions is the sonar interrupt handler, called *ih_sonar*. This program is written in assembler code and is the interface between the sonar hardware and the sonar software system. When the hardware has data ready and generates an interrupt, *ih_sonar* first saves the state of the processor. It then places the sonar data on the stack, determines which of the sonars are over-ranged and places that data on the stack, places data about the robot's position on the stack and calls *serve_sonar.* When *serve_sonar* returns control to *ih_sonar*, the processor is returned to the state it was in before the interrupt and the interrupt processing is complete.

Serve_sonar serves two major purposes. First, it loads the correct range data into the *sonar_table* along with the robot's positional data and sets a flag indicating that the data for that sonar has been updated. Second, it examines a number of flags and determines which functions must be performed on which data. The scheduling which occurs here is the first step in paring the processor's work load to the minimum amount necessary to accomplish the user's desires.

If *serve_sonar* determines that global coordinates are required for a given sonar, it calls the *calculate_global* function which will perform the calculations and load the coordinates into the *sonar_table* for that sonar. Next, it determines whether the user desires line segments to be derived from the data. If so, it calls the *linear_fitting* function. *Linear_fitting* in turn uses several subroutines, including *start_segment* to gather the initial points for the line segment, *add_to_line* to add data points to the line segment after it's initial formation, and *end_segment* to terminate the line segment when deemed necessary. The *build_list* function is used to add the completed segment to the proper list, and the *reset_accumulators* function is used when necessary to reset the summations for the linear fitting algorithm.

When the various data conversions are complete for a given sonar, *serve_sonar* determines whether the user desired for any of the data to be logged, and if so calls the *log_data* function which extracts the correct data and places it in the appropriate array. At the completion of the mission, the *host_xfer* function is used to transmit the data as a single string to the host computer.

The reader will note one additional function in the sonar.c file. That is *msbn*, which is the target of an interrupt handler *ih_msbn*. *Msbn* is the location of a future precise navigation system, and as such will doubtless be part of it's own source file. For the interim, *msbn* is included in sonar.c as a placeholder for *ih_msbn*.

Figures 6.1a, 6.1b and 6.1c on the following pages give a graphic representation of the sonar system functions and their basic interrelations. The background functions are arranged on the left side of the diagram and the user functions on the right side. Key data

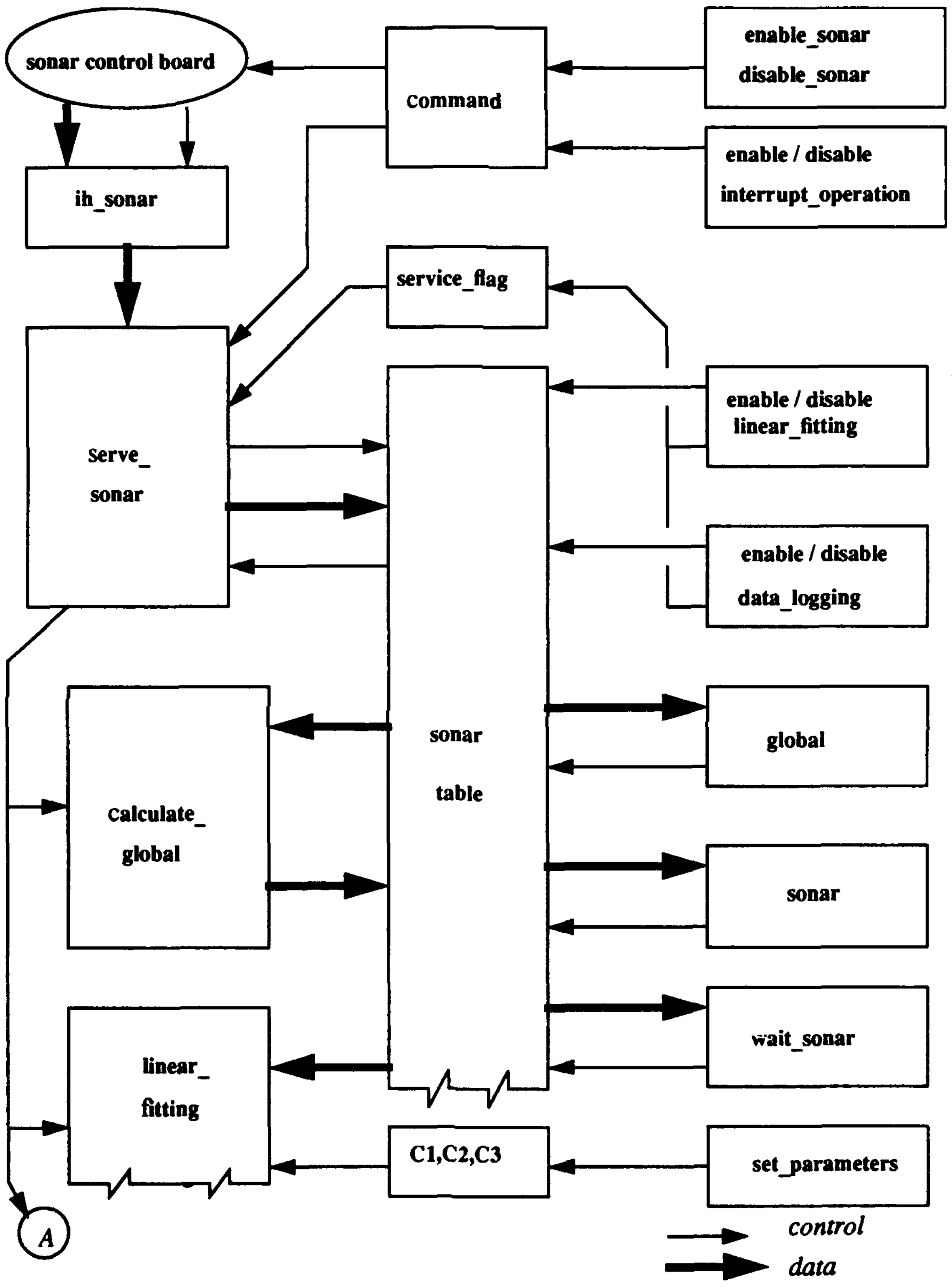

Figure 6.1a. Sonar System Block Diagram

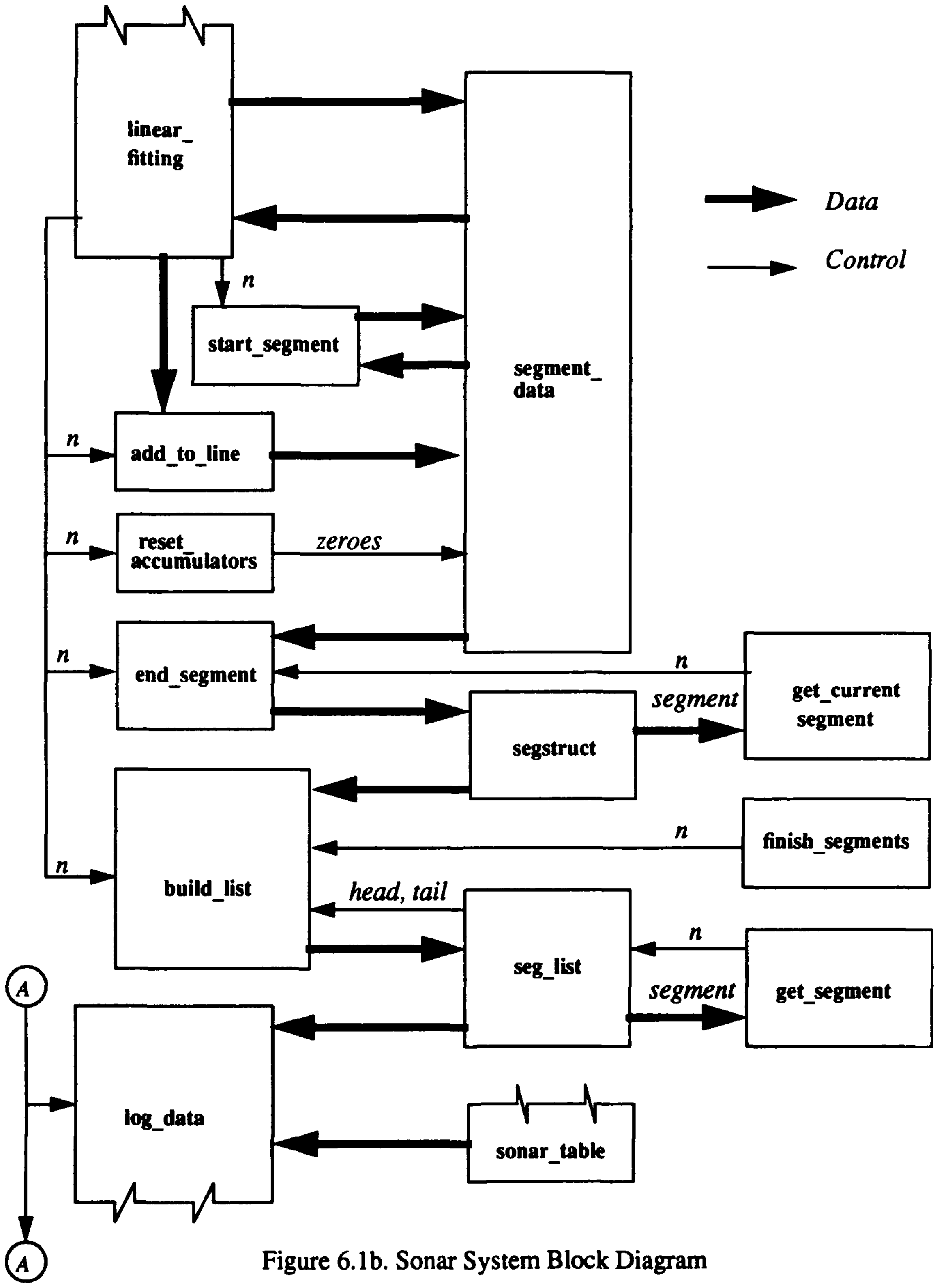

Figure 6.1b. Sonar System Block Diagram

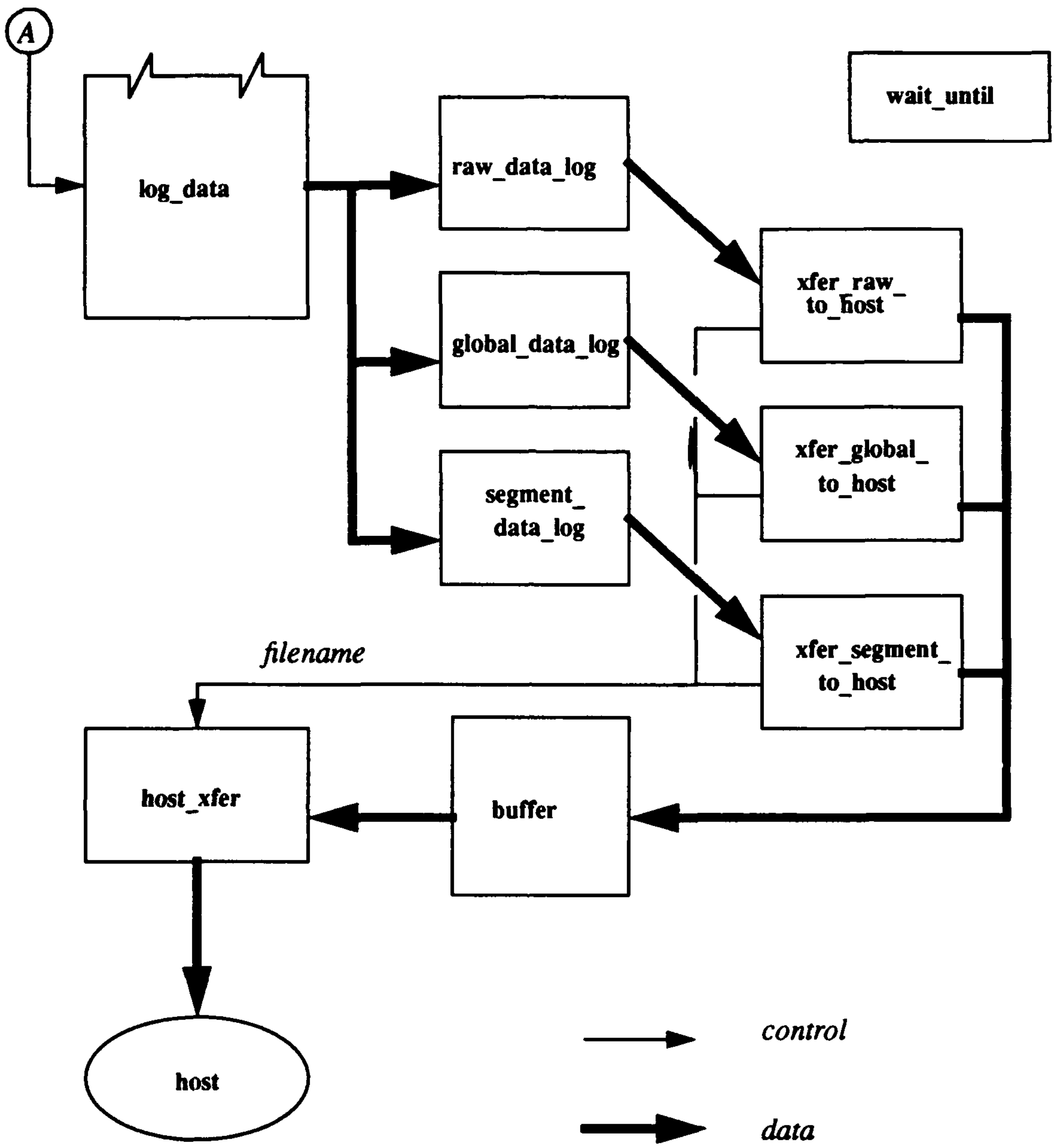

Figure 6.1c. Sonar System Block Diagram

structures are displayed in the middle. Sonar data flow is represented by heavy lines, while control and selection values are represented by the light lines.

C. Data Structures

The following are additional data structures which are hidden from the user. Knowledge of these data structures is necessary only to understand the functioning of the background system.

1. Segment Data

Structure:

```
typedef struct{
int       n,
          rst,
          sgmp;
double    initx[3],
          inity[3],
          sgmx,
          sgmy,
          sgmx2,
          sgmy2,
          sgmxy,
          sgm_delta_sq,
          theta,
          r,
          d_major,
          d_minor,
          startx,
          starty,
          endx,
          endy;
          } CUR_DATA;
CUR_DATA segment_data[16];
```

Description: the operation of the linear fitting algorithm is dependant on the accumulation of certain sums. Since the processor is not dedicated to the linear fitting process for a given sonar, we must be able to save these summations in some interim state between iterations of the linear fitting algorithm. This is the purpose of the sgmx, sgmy, sgmx2, sgmy2, sgmxy and sgm_delta_sq items in CUR_DATA. In addition, we save the uncorrected starting position and ending position of the segment (startx, starty, endx and endy) and some parameters of the segment that are updated for every iteration (theta, r, dmajor and dminor). The integers represent the sonar number (n), the number of data points thus far included in the line segment (sgmp) and a number indicating which state the linear fitting algorithm should return to for the next iteration of this line segment (rst). The structures are arranged as a sixteen wide array indexed by sonar number.

2. Miscellany

There are some other variables and pointers mentioned that deserve some explanation. These include:

-service_flag	an integer variable used to track the number of enabled processes. If service_flag = 0 then no processes (global conversion, linear fitting, etc.) are active and the *serve_sonar* function can short circuit a large portion of code.
-enabled_sonars[16]	indexed by sonar number. A value of 1 at a position would indicate that sonar is in use.
-command_ptr	a pointer to the command register on the sonar control board. This is a write only register; the value of the command register may not be read.
-enabled	a variable that contains the current value of the command register. This is where the contents of the command register must be read.
-status_ptr	a pointer to the status register of the sonar control board. This is a read only register and is at the same address as the command register. Therefore, if an attempt is made to read the command register vice enabled, the contents of the status register is what will be returned.
-BIM_ptr	a pointer to the BIM control register. The BIM is the Bus Interface Module on the VME motherboard which carries the sonar control board.

-group_array[4][4]	this array maps sonar numbers to groups.
-C1, C2, C3	constants for linear fitting algorithm. Declared in mml.h, initialized in main.c.

D. Background Function Synopsis

While the *user sonar functions* are those the user would employ in his/her program, the *background sonar functions* are those functions the system employs to control the sonar system and achieve the users objectives. Code for the interrupt handler is provided in Appendix B, and code for the "C" functions is provided in Appendix A. A brief synopsis of the functions are provided here.

1. Interrupt Handler

Syntax: ih_sonar

Description: this is an assembly code program which resides in interrupt.s. When the sonar control board's interrupt is acknowledged by the processor, the BIM provides a vector to the starting address of this program. This program then accomplishes the following tasks:

- saves the state of the processor and coprocessor
- loads the sonar group number onto the stack
- loads the sonar data registers onto the stack
- composes the overflow word and loads it onto the stack
- loads robot's current x,y,t onto the stack
- calls the *serve_sonar* routine
- upon return, restores the processor and coprocessor to their previous state

2. Serve Sonar

Syntax:
```
void serve_sonar(x,y,t,ovfl,data4,data3,data2,data1,group)
double x, y, t;
int ovfl, data4, data3, data2, data1, group;
```

Description: this function is the "central command" for the control of all sonar related functions. It is linked with the ih_sonar routine and loads sonar data to the sonar_table from there. It then examines the various control flags to determine which activities the user wishes to take place and calls the appropriate functions. This function is invoked only when a sonar group is enabled, causing the sonar control board to generate interrupts. If one sonar group is enabled, this function is invoked approximately every 25 milliseconds.

3. Calculate Global

Syntax:	void calculate_global(n) int n;
Description:	calculates the global x and y coordinates of the point located by sonar n. Uses the range value for sonar n, the position and orientation of the robot at the time the range was acquired, and the dislocation of the sonar from the center of the robot. All of this data is located in sonar_table[n]. Function first permutes the posture of the robot to find the global posture of the selected sonar, using the phi, offset and axis values from the sonar table. The global x and y position of the sonar is then translated along the axis of the sonar beam the distance d, resulting in the global x and y values for the echoing point. These are stored as gx and gy in the sonar table. See Figure 6.2.

4. Linear Fitting

Syntax:	void linear_fitting(n) int n;
Description:	controls the fitting of global coordinate data to straight line segments. First collects three data points to initialize a line segment (see *start_segment).* After the segment is established the procedure tests each subsequent point for it's acceptability prior to adding it to the segment (see *add_to_line).* After including the data point the segment is tested to ensure the entire set of data points is "linear enough". If two out of three points are unacceptable or if the set of points fails linearity checks the line segment is ended and a new one started (see *end_segment*). The completed line segment data is recorded in a data structure called *segment* and the segment is added to a two dimensional array called *seg_list*. Seg_list is indexed by sonar number and pointers to the oldest and newest segments stored at that sonar number. Function uses a multivalue flag *rst* to track the status of the line segment and the three latest points being fitted. See the Linear Feature Extraction chapter for more details

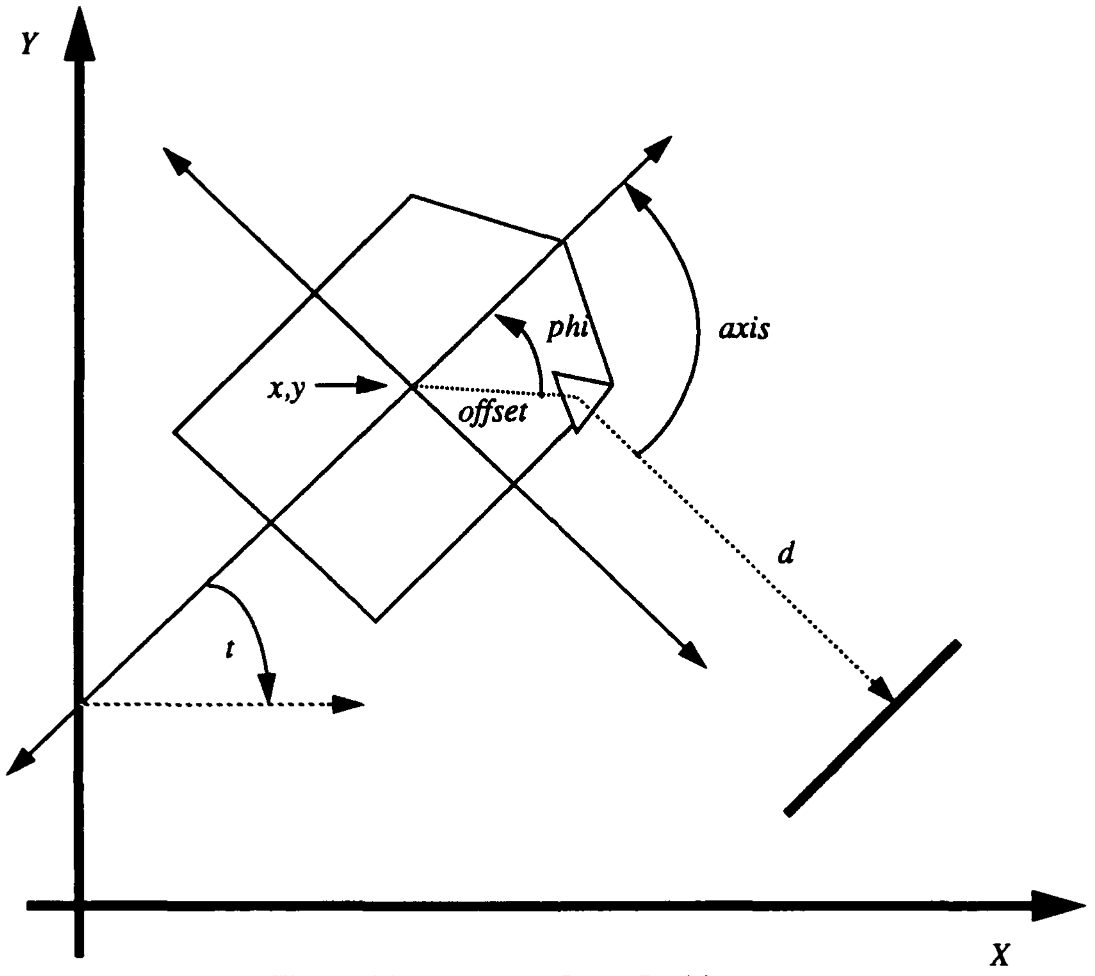

Figure 6.2. Robot and Sonar Position Values.

5. Start Segment

Syntax:	void start_segment(n) int n;
Description:	establishes a new line segment with the three data points contained in segment_data[n].init(x and y). It writes the appropriate data to the interim values in segment_data[n].

6. Add To Line

Syntax:	void add_to_line(n,x,y)
	int n;
	double x, y;

Description: calculates new interim data for the line segment and stores it in segment_data[n]. Input parameters x and y are the global x and y coordinates of the point being added. The function also calculates the thinness ratio for the segment and updates segment parameters if the ratio is satisfactory, otherwise it sets flag *rst* to indicate that the segment failed thinness and must be ended.

7. End Segment

Syntax:
```
LINE_SEG *end_segment(n)
int n;
```

Description: calculates the true endpoints of the line segment and it's length. The data is written into a temporary buffer and a pointer to that buffer is returned.

8. Reset Accumulators

Syntax:
```
void reset_accumulators(n)
int n;
```

Description: resets the accumulative values in segment_data[n] (sgmx, sgmy, sgmx2, sgmy2, sgmxy) to zero.

9. Build List

Syntax:
```
void build_list(ptr,n)
int n;
LINE_SEG *ptr;
```

Description: stores the segment that ptr points to in the array seg_list at the next position (pointed at by the tail pointer) and then updates the tail pointer. If the tail pointer rolls around to the location of the head pointer (seg_list is essentially 16 circular queues) it will also move the head pointer up one position. This destroys the oldest segment in the queue and keeps the five newest.

10. Log Data

Syntax:
```
void log_data(n,type,filenumber,i)
int n, filenumber, type, i;
```

Description: causes data to be written to a file. The filenumber specifies to which of the four files (0, 1, 2, 3) for a given data type the data will be written to. The value of i is used to index the seg_list array for storing line segments.

11. Host Xfer

Syntax:	void host_xfer(buffer,filename)
	char *buffer;
	char *filename;

Description:	transfers a data string to the host.

VII. TEST RESULTS AND CONCLUSIONS

Testing was conducted in two phases. In the first, or static, phase the robot was supported on blocks and the operating parameters of the sonar were tested. In the second, or dynamic, phase the locomotion functions were called into play. The robot was first allowed to run a mission while on the blocks, to ensure the interoperability of the sonar and locomotion systems. The robot was then run in the hallway of Spanagel Hall to gather real world data for download to the host system, a Sun 3/60.

A. Static Testing

The first trial program (Appendix C.1) simply asked the user for a sonar number and then displayed the range returned by that sonar repetitively, with a delay set by a number of wait_sonar commands. The second program (Appendix C.2) was only slightly more complex and allowed for the user to enable or disable multiple sonars. Using these programs, we were able to determine that each of the sonars did indeed work and return a correct range, and we were able to determine the beamwidth of the sonar.

1. Sonar Operability

Table 7.1 lists the results of the individual tests of the sonars at a range of approximately 500 millimeters. The a and b values are the distances from the receiver transducer and the transmitting transducer to the target, respectively, as shown in figure 7.1.While we expected to find ranges equal to the average of these two values, the actual ranges more closely followed the target - receiver distance. The average error was +18.95 mm., placing the average sonar centroid very near the location of the receiver transducer. The greatest error noted was +32.5 mm., or 6.5% of the indicated range of 504 mm. range for sonar 9.

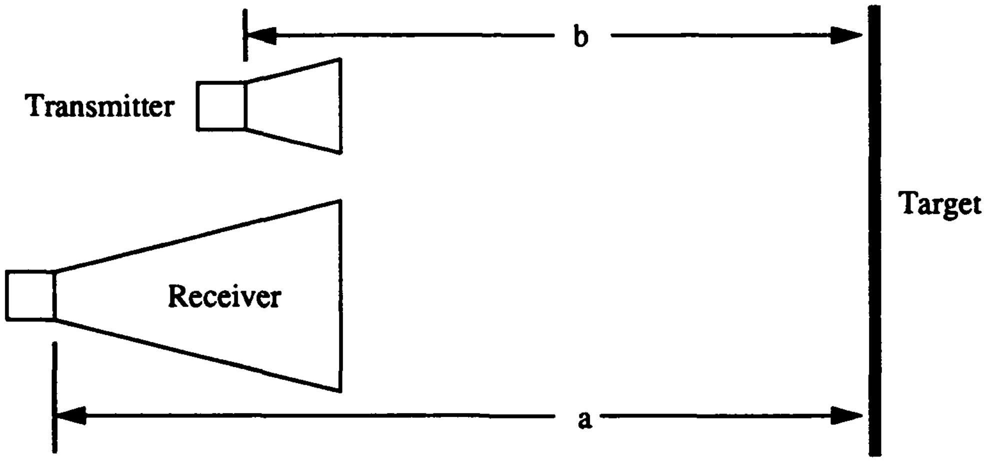

Figure 7.1. Transmitter - Receiver Offset

Table 1: Sonar Ranging Test Results

Sonar	a (mm)	b (mm)	range (mm)
0	518	478	510
1	518	480	513
2	520	484	514
3	513	474	515
4	495	455	503
5	492	454	479
6	495	455	499
7	488	449	482
8	505	465	499
9	500	463	504
10	488	449	486
11	495	452	506

2. Effective Beamwidth

In the next portion of the static testing, we used a movable target wall to experimentally determine the *effective* beamwidth of the sonar. The theoretical beamwidth of a sonic transducer in the far field may be calculated as

$$\theta = (1.22\lambda)/D \qquad \text{(Eq 7.1)}$$

where θ is in radians, λ is the acoustic wavelength and D is the transducer diameter. For our operating frequency of 40 KHz and with a transducer diameter of 1.5 cm., we calculate a theoretical beamwidth of 0.697 radians, or 40 degrees. We expect this value to be reduced somewhat by the beam shaping cones placed around each transducer. Also, we note that we are actually measuring conditions under which enough sonic energy is returned to the receiver to exceed the threshold value necessary to register a return pulse. This will form an "effective beamwidth" which is *much less* than the theoretical beamwidth and dependent on transmitter signal strength, receiver sensitivity, air conditions and the nature of the reflecting surface. Out test setup is depicted in Figure 7.2.

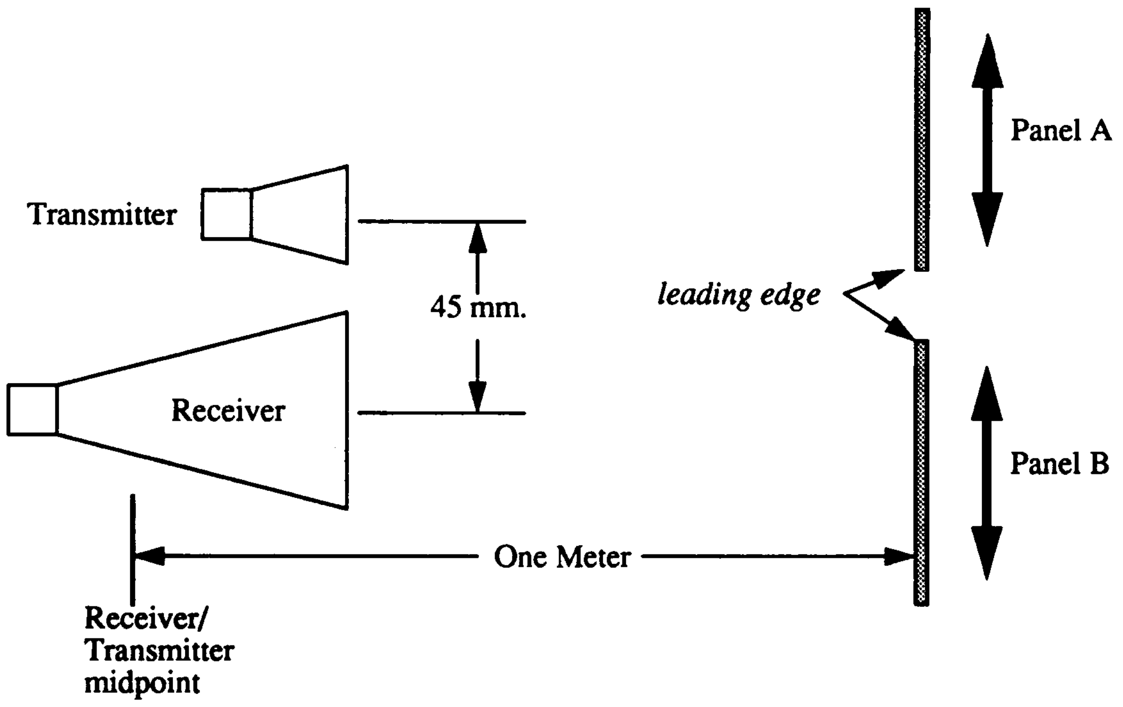

Figure 7.2. Sonar Beam Test Setup

The leading edges of the two panels were insinuated into the sonar beam, first individually and then simultaneously, to determine the point at which usable sonar information was returned. For our purposes, usable sonar information was interpreted to mean at least 50% of the returns actually provide a range, rather than an over-range (no return) signal.

When the leading edge of panel A was inserted to the midpoint of the receiver's axis, 50% of the sonar pulses produced a return. Those returns were of range 103 centimeters. Similarly, the insertion of panel B to the midpoint of the receiver's axis produced a 50% return rate with a range of 103 centimeters. With both panels inserted to the midpoint of their respective transducers axis, a 100% return rate was achieved with a range value of 100 centimeters. This variation can be explained by examining the transmitted and received pulse's waveshapes, shown in Figure 7.3.

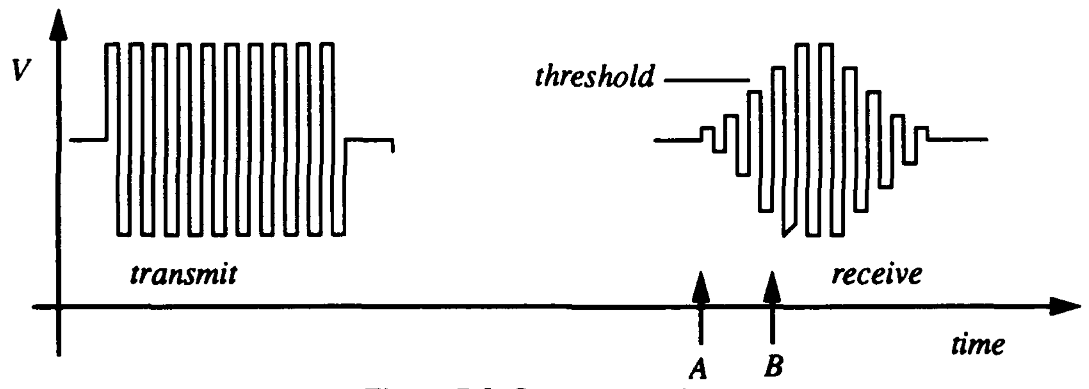

Figure 7.3. Sonar Waveshapes

While the return shown actually begins at point A, and the elapsed time at point A would produce an accurate range, the sonar does not detect the return until point B, when the return pulse exceeds the receiver threshold. If we presume that the return pulse is barely sufficient to exceed the threshold, as it must be if we achieving only a 50% return rate, and knowing that our transmit pulse width is 500 microseconds, we may calculate that an additional 250 microseconds or 8.58 centimeters are added to the round trip distance. This maximum delay would be 4.29 centimeters of actual range and is, indeed, the value indicated when the return rate drops to 10% or less as the panel is withdrawn.

The 100 centimeter range returned with both panels inserted to the axis midpoints indicates that the return signal is exceeding the threshold much earlier than it was with a single panel. This would be expected, as the reflecting surface is increased in area thus causing more sonic energy to be returned to the receiver. In keeping with this theory, we find that if the two panels are pushed completely together, forming a continuous reflecting surface, the indicated range drops to 99 centimeters.

As can be seen from Figure 7.2, the transducer axes are displaced from the center of the transducer pair by 22.5 millimeters. This distance, at 1 meter range, equates to an angle of only 1.3 degrees, thus forming an effective beamwidth at 1 meter of only 2.6 degrees. This very narrow beamwidth will permit precise location of surface edges when applied in an orthogonal world,and also makes the sonar very sensitive to surface orientation. If the surface is not within a very few degrees of perpendicular to the sonar beam's axis, the reflected pulse will not fall within the sonar's receive cone and will be lost.

B. Dynamic Testing

Initial testing of the sonar system in combination with the locomotion system were conducted with the robot supported on wooden blocks. In addition to ensuring the two systems would run together concurrently, these tests allowed the author to learn how to write user programs that actually accomplished the intended goal. Because the sequential locomotion functions are placed on a queue as they are encountered in the user program, the user must learn when to delay further execution of the user program so that the robot may physically "catch up" to the program. Learning these techniques with the robot simply spinning it's wheels rather than lurching about the hallway was a great timesaver.

Once the programming techniques were learned, a simple program was written that had the robot proceed for 6 meters while recording global position data and deriving and recording line segments descriptive of that data (Appendix C.3). The venue for the tests was Spanagel Hall, fifth deck at Naval Postgraduate School and included two doorways that served as markers for the output data (Figure 7.4). The robot was started at the same point for each data run, and two graphs display the results of each run. The left hand graph is a display of the individual data points in global x and y coordinates, the right hand graph is a display of the line segments derived by the robot to fit the data points. The robot proceeded along a simple DR (dead reckoned) path to collect the data, the direction being determined by the robot's initial heading. Deviation from a path parallel to the measured wall accounts for the data's tendency to "lean" one way or the other from vertical.

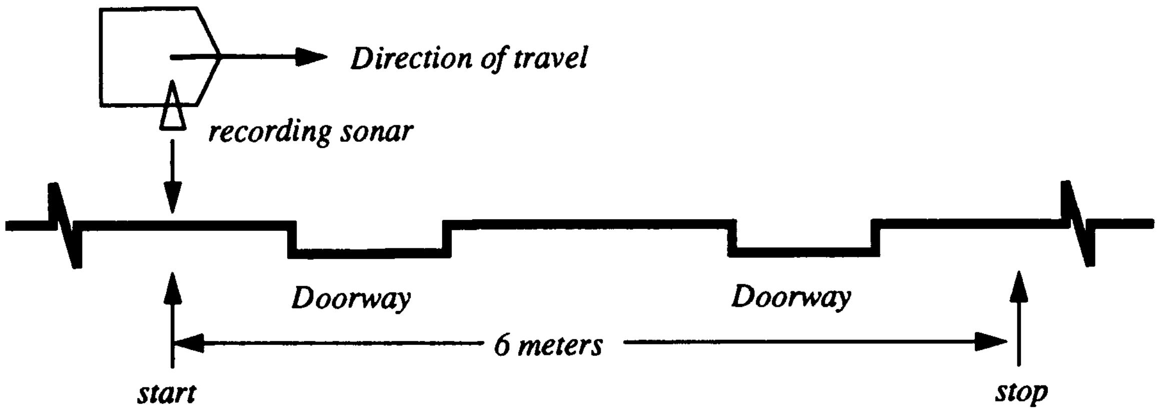

Figure 7.4. Sonar Dynamic Testing Venue

1. Test One

In this first test, the linear fitting parameters C1, C2 and C3 were set at 3.0, 5.0 and 0.1 respectively. This means that individual data points would be acceptable for inclusion in a line segment if they were displaced from the line segment by less than the maximum of either 3.0 times *sigma* or 5.0 centimeters. The line segment will be terminated if it's thinness ratio exceeds 0.1. The data interval for this test, and for the ensuing tests, is set to 5. As can be seen in Figure 7.5, the global coordinate pairs returned by the sonar accurately describe the wall and doorways being mapped. The linear fitting of the line segments, however, leaves much to be desired.

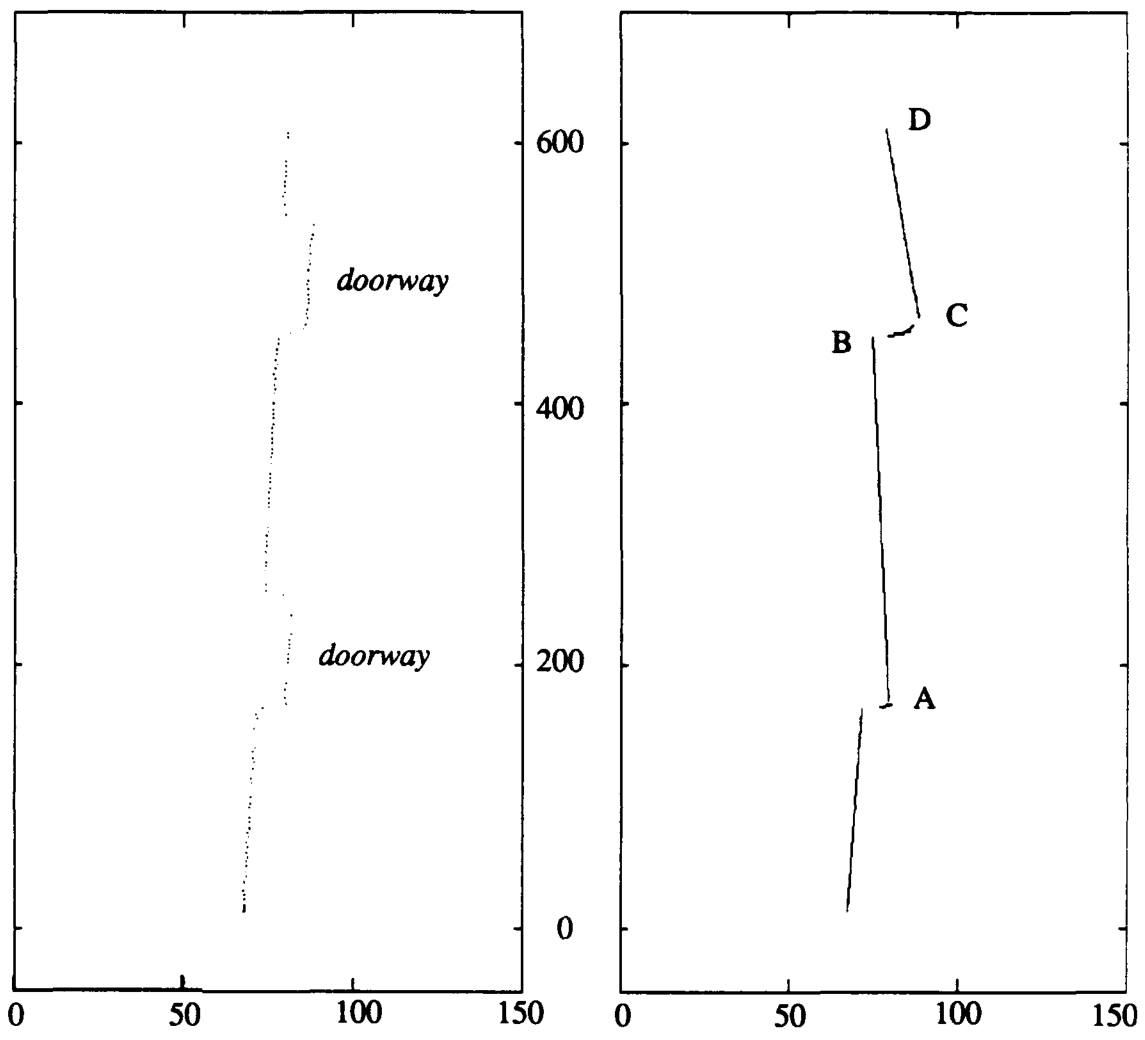

Figure 7.5. Sonar Dynamic Test One

Line segment AB incorporates two features, the doorway and the ensuing wall. Line segment CD likewise incorporates a doorway and the ensuing wall. These segments must be broken up into their two components in order to be useful.

2. Test Two

The offset from the walls to the doorway surfaces in our test area is approximately 9 centimeters. We judged that the C2 parameter of 5.0 was too large to allow clear definition of features of that size, so for the second test run we lowered C2 from 5.0 to 2.0 centimeters. The results of that test run are shown in Figure 7.6.

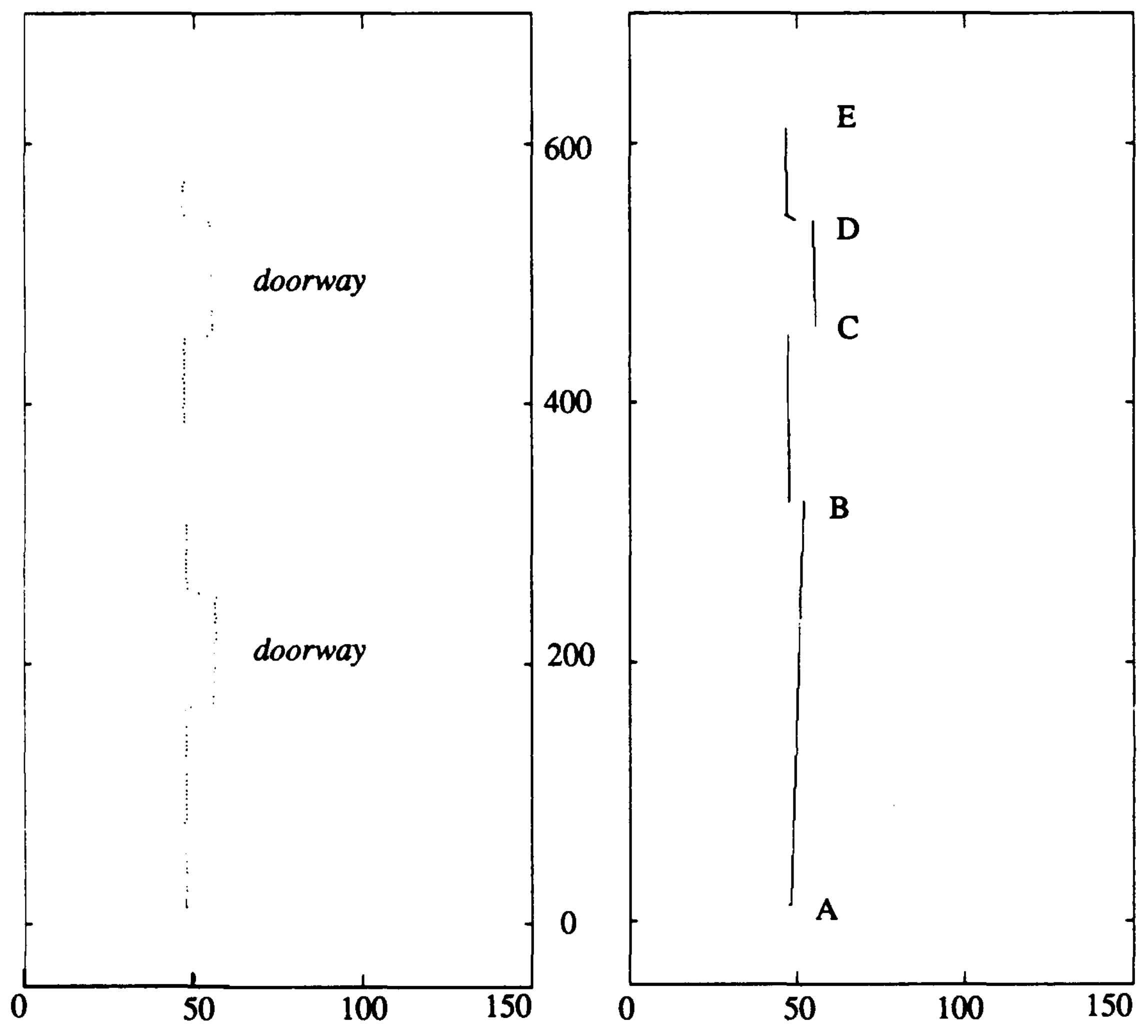

Figure 7.6. Sonar Dynamic Test Two

In this case, line segments BC, CD and DE accurately represent the latter half of the data run. The first half, however, incorporates the first doorway and both the leading and trailing wall portions in one line segment, AB. Another adjustment is necessary.

3. Test Three

For the third test we opted to change parameter C1, the sigma multiplier, from 3.0 down to 2.0. Parameter C2 remains at 2.0 and parameter C3 at 0.1. The results are shown in Figure 7.7.

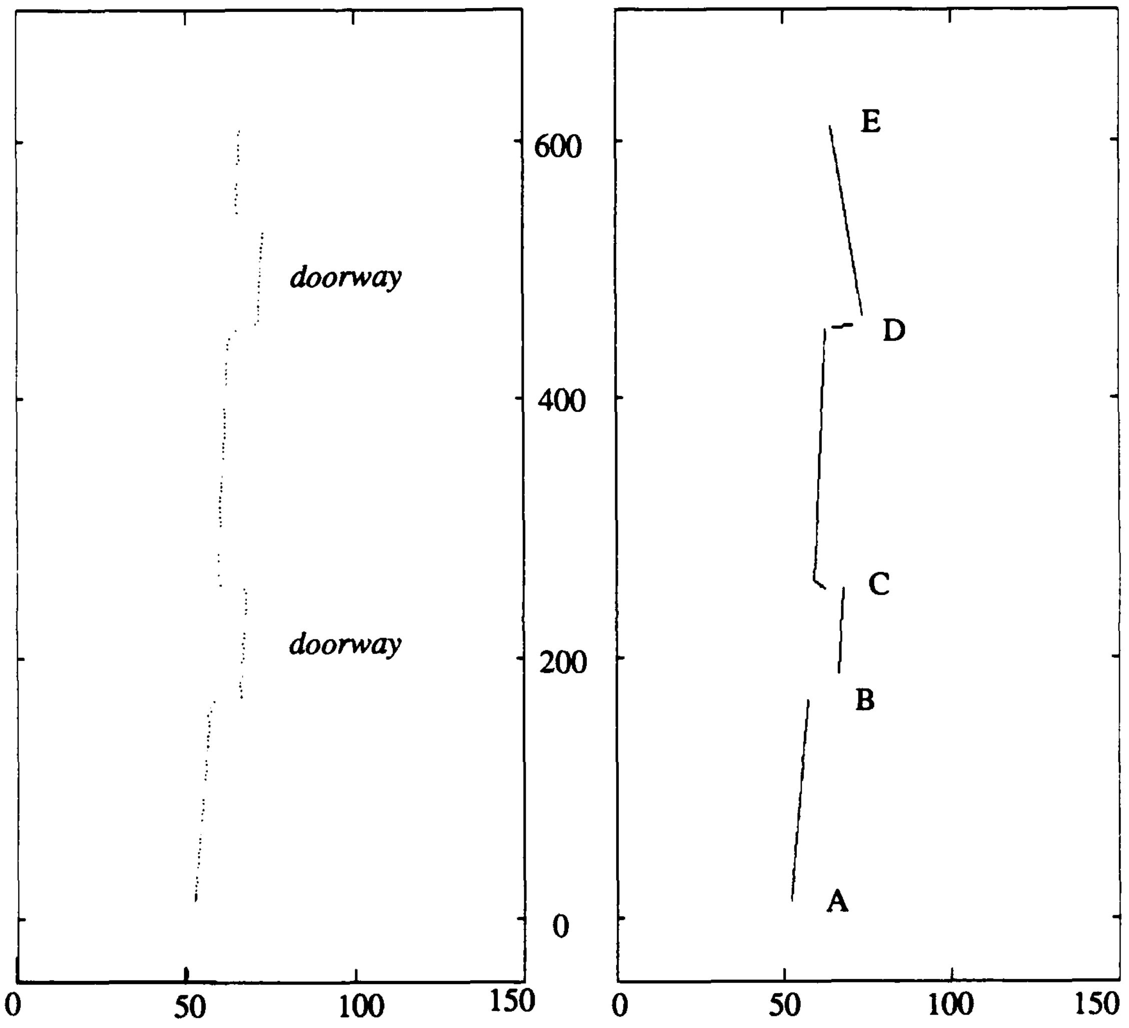

Figure 7.7. Sonar Dynamic Test Three

Results from this test are not substantively different from test two, with the exception that the incorrect line segment, DE, has endpoints which coincide with breaks in the pattern of data points. This is still unsatisfactory.

4. Test Four

For the next iteration of the test we turn our attention to parameter C3. Parameters C1 and C2 are already adjusted as small as we judge is practical, while the linearity of the data points suggests that a thinness factor of one in ten may be overly generous. We adjusted the parameter C3 down to 0.08, so that our parameters are now 2.0, 2.0 and 0.08 for C1, C2 and C3 respectively. The results are shown in Figure 7.8.

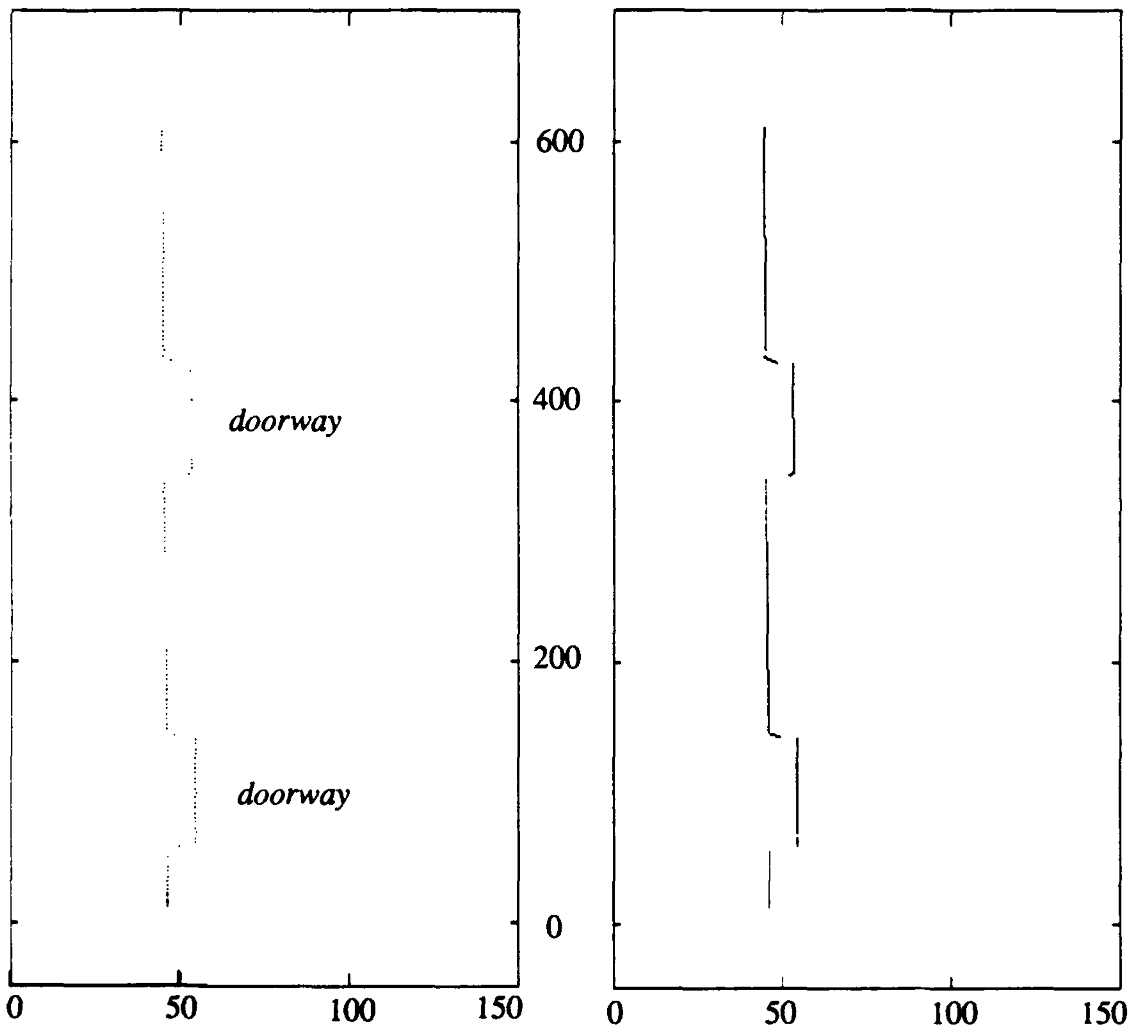

Figure 7.8. Sonar Dynamic Test Four

This combination of parameters for the linear fitting algorithm has produced perfect results. The doorways are clearly defined and the length and orientation of the segments match the sensed environment exactly.

C. Conclusions

The sonar system designed for and implemented aboard Yamabico achieves the goals set for it. Specifically, it:

- provides basic sonar data on demand
- extracts linear features from the data
- adapts to user demands to minimize processor time
- at the lowest level, operates completely autonomously
- has reduced hardware complexity to a great degree
- provides a friendly interface to the user

There are still some shortcomings to the system, however, that will require further work to resolve. Most notable are the problems with the effective beamwidth of the individual sonars. The extremely narrow beamwidth leaves large areas of the sonar azimuth uncovered, thus eliminating any possibility of detecting objects that appear in the uncovered areas between beams. The most probable fix for this problem is an increase in sonar transmitting power, although a different transducer may also effect a significant improvement.

Another significant problem is the relatively small amount of memory available to the processor (one megabyte). Logging sonar data of large areas for later download, or creating onboard maps of the environment, will require substantially more memory than is currently available. While volatile RAM may be increased to accommodate this need for additional memory, it is the author's opinion that conversion to an onboard processing system with it's own operating system supporting some sort of mass storage would better serve current and future computational requirements. The incorporation of higher level processes, such as a navigator to utilize the sonar data now generated or the future installation of a vision system, will certainly require more onboard computing power.

APPENDIX A - CODE FOR SONAR "C" FUNCTIONS

```
/* sonar.c */
/* ultrasonic rangefinder functions */
#include "mml.h"

#define print_flex(x,y) y = putstr(" ", putstr(rtoae((double) (x), tmpstr, 4), y))
#define nl_flex(x) x = putstr("\n", x)

/*declaration of functions and return values*/

extern double sonar();
extern void enable_sonar();
extern void disable_sonar();
extern void msbn();
extern double wait_sonar();
extern posit global();
extern void enable_linear_fitting();
extern void disable_linear_fitting();
extern void enable_data_logging();
extern void disable_data_logging();
extern void serve_sonar();
extern LINE_SEG *get_segment();
extern LINE_SEG *get_current_segment();
extern void set_parameters();
extern void enable_interrupt_operation();
extern void disable_interrupt_operation();
extern void calculate_global();
extern void linear_fitting();
extern void start_segment();
extern void add_to_line();
extern LINE_SEG *end_segment();
extern void reset_accumulators();
extern void build_list();
extern void log_data();
extern void set_log_interval();
extern void wait_until();
extern void xfer_raw_to_host();
extern void xfer_global_to_host();
extern void xfer_segment_to_host();
extern void host_xfer();
extern void finish_segments();
```

```
/**********************************************************************/
/*
/* Procedure:  sonar(n)
/*
/* Description:  returns the distance (in centimeters) sensed by the
/*     n_th ultrasonic sensor.  If no echo is received, then a -1 is
/*     returned.  If the distance is less than 10 cm, then a 0 is
/*     returned.
/*
/**********************************************************************/

double sonar(n)
int n;
{
return (sonar_table[n].d);
}

/**********************************************************************/
/*
/* Procedure:  enable_sonar(n)
/*
/* Description:  enables the sonar group that contains sonar n, which
/*     causes all the sonars in that group to echo-range and write data
,*     to the data registers on the sonar control board. Marks the n'th
/*     position of the enabled_sonars array to track which sonars are
/*     enabled.
/*
/**********************************************************************/

void enable_sonar(n)
int n;
{

int i;
i = imaskoff();
      enabled_sonars[n] = 1;
      switch (n)
      {
case 0:
      case 2:
      case 5:
      case 7:
        enabled = enabled | 0x01;
        break;
      case 1:
      case 3:
      case 4:
      case 6:
        enabled = enabled | 0x02;
        break;
      case 8:
      case 9:
      case 10:
      case 11:
        enabled = enabled | 0x04;
        break;
      case 12:
      case 13:
      case 14:
      case 15:
        enabled = enabled | 0x08;
        break;
      }
      *command_ptr = enabled;
imaskon(i);
}

/**********************************************************************/
/*
/* Procedure:  disable_sonar(n)
/*
/* Description:  removes the sonar n from the enabled_sonars list. If
/*     sonar n is the only enabled sonar from it's group, then the
/*     group is disabled as well and will stop echo ranging. This has
/*     benefit of shortening the ping interval for groups that remain
/*     enabled.
/*
/**********************************************************************/

void disable_sonar(n)
int n;
{
int i,c;
      char mask;

i = imaskoff();
      enabled_sonars[n] = 0;
      switch (n)
```

```
          {
          case 0:
          case 2:
          case 5:
          case 7:
           c = enabled_sonars[0] + enabled_sonars[2] +
             enabled_sonars[5] + enabled_sonars[7];
           if (c == 0) enabled = enabled & 0xfe;
           break;
          case 1:
          case 3:
          case 4:
          case 6:
           c = enabled_sonars[1] + enabled_sonars[3] +
             enabled_sonars[4] + enabled_sonars[6];
           if (c == 0) enabled = enabled & 0xfd;
           break;
          case 8:
          case 9:
          case 10:
          case 11:
           c = enabled_sonars[8] + enabled_sonars[9] +
             enabled_sonars[10] + enabled_sonars[11];
           if (c == 0) enabled = enabled & 0xfb;
           break;
          case 12:
          case 13:
          case 14:
          case 15:
           c = enabled_sonars[12] + enabled_sonars[13] +
             enabled_sonars[14] + enabled_sonars[15];
           if (c == 0) enabled = enabled & 0xf7;
           break;
          }
         *command_ptr = enabled;
    imaskon(i);
    }


/************************************************************************/
/* Procedure:  msbn
/*
/* Description:  called every 5 ms, this routine drives the precise
/*    navigation system.
```

```
/*
/**********************************************************************/

void msbn()
{
}

/**********************************************************************/
/*
/* Procedure: wait_sonar(n)
/*
/* Description:  waits in a loop until new data is available for
/*     sonar n.
/*
/**********************************************************************/

double wait_sonar(n)
int n;
{
     int a = 0;

     sonar_table[n].update = 0;
     while (sonar_table[n].update == 0);
     return(sonar_table[n].d);
}

/**********************************************************************/
/*
/* Procedure: global(n)
/*
/* Description: returns a structure of type posit containing the global
/*     x and y coordinates of the position of the last sonar return.
/*
/**********************************************************************/

posit global(n)
int n;
{
  posit answer;

  if (sonar_table[n].global == 0) calculate_global(n);
  answer.gx = sonar_table[n].gx;
  answer.gy = sonar_table[n].gy;
  answer.psi = sonar_table[n].t + sonar_table[n].axis;
  return answer;
}

/************************************************************************/
/*
/* Procedure: enable_linear_fitting(n)
/*
/* Description:  causes the background system to gather data points
/*     from sonar n and form them into line segments as governed by
/*     the linear fitting algorithm. Increments service_flag.
/*
/************************************************************************/

void enable_linear_fitting(n)
int n;
{
      sonar_table[n].fitting = 1;
      sonar_table[n].global = 1;
      ++service_flag;
}

/************************************************************************/
/*
/* Procedure:  disable_linear_fitting(n)
/*
/* Description:  causes background system to cease forming line
/*     segments for sonar n. Decrements the service_flag.
/*     Will also disable the calculation of global coordinates for
/*     that sonar if data logging of global data is not enabled.
/*
/************************************************************************/

void disable_linear_fitting(n)
int n;
{
  sonar_table[n].fitting = 0;
  if (sonar_table[n].filetype[1] == 0) sonar_table[n].global = 0;
  --service_flag;
}

/************************************************************************/
/*
/* Procedure: enable_data_logging(n,filetype,filenumber)
/*
```

```
/* Description: causes the background system to log data for sonar (n)
/*    to a file (filenumber). The data to be logged is specified by an
/*    integer flag (filetype). A value of 0 for filetype will cause raw
/*    sonar data to be saved, 1 will save global x and y, and 2 will
/*    save line segments. The filenumber may range between 0 and 3 for
/*    each of the three types, providing up to 12 data files. Example:
/*        enable_data_logging(4,1,0);
/*    will cause raw data from sonar #4 to be saved to file 0, while:
/*        enable_data_logging(7,2,0);
/*    will cause segments for sonar #7 to be saved to file 0.
/*    Function increments the service_flag.
/*
/**********************************************************************/

void enable_data_logging(n,filetype,filenumber)
int n, filetype, filenumber;
{
  if (filetype == 1) sonar_table[n].global = 1;
  sonar_table[n].filetype[filetype] = 1;
  sonar_table[n].filenumber[filetype] = filenumber;
  ++service_flag;
}

/**********************************************************************/
/*
/* Procedure: disable_data_logging(n,filetype)
/*
/* Description: causes the background system to cease logging data of a
/*    given filetype for a sonar n. Decrements the service_flag.
/*
/**********************************************************************/

void disable_data_logging(n,filetype)
int n,filetype;
{
  if ((filetype == 1) && (sonar_table[n].fitting == 0)) sonar_table[n].global = 0;
  sonar_table[n].filetype[filetype] = 0;
  --service_flag;
}

/**********************************************************************/
/*
/* Procedure: serve_sonar(x,y,t,ovfl,data1,data2,data3,data4,group)
/*
```

```
/* Description:  this procedure is the "central command" for the
/*    control of all sonar related functions. It is linked with
/*    the ih_sonar routine and loads sonar data to the sonar_table
/*    from there. It then examines the various control flags in the
/*    sonar_table to determine which activities the user wishes to
/*    take place, and calls the appropriate functions. This procedure
/*    is invoked approximately every thirty milliseconds by an
/*    interrupt from the sonar control board.
/*
/*************************************************************************/

void serve_sonar(x,y,t,ovfl,data4,data3,data2,data1,group)
double x,y,t;
int ovfl,data4,data3,data2,data1,group;
{

    int i;
    int data[4];
    int ovfl_mask = 8;

    data[0] = data1;
    data[1] = data2;
    data[2] = data3;
    data[3] = data4;

    for (i = 0; i < 4; i++, ovfl_mask /= 2)
 {
      if (ovfl_mask & ovfl)
        sonar_table[group_array[group][i]].d = -1.0;
      else if (data[i] < 100)
            sonar_table[group_array[group][i]].d = 0.0;
          else sonar_table[group_array[group][i]].d = (float)data[i] / 10.0;
      sonar_table[group_array[group][i]].x = x;
      sonar_table[group_array[group][i]].y = y;
      sonar_table[group_array[group][i]].t = t;
      sonar_table[group_array[group][i]].update = 1;
 }


    if (service_flag != 0)
{
     for (i = 0; i < 16; i++)
 {
       if (sonar_table[i].update == 1)
    {
        if (sonar_table[i].global == 1)
          calculate_global(i);
        if (sonar_table[i].fitting == 1)
          linear_fitting(i);
        if (sonar_table[i].filetype[0] == 1)
          log_data(i,1,sonar_table[i].filenumber[0],0);
        if (sonar_table[i].filetype[1] == 1)
          log_data(i,2,sonar_table[i].filenumber[1],0);
        }
       sonar_table[i].update = 0;
      }
     }
}

/************************************************************************/
/*
/* Procedure: get_segment(n)
/*
/* Description:  returns a pointer to the oldest segment on the linked
/*     list of segments for sonar n; i.e. the record at the head
/*     of the linked list. It is destructive, thus subsequent calls
/*     will return subsequent segments until the list is empty.  This is
/*     accomplished by first copying the contents of the head record
/*     into a temporary record called segstruct and then freeing the
/*     allocated memory for the head record. The pointer returned is
/*     actually a pointer to this temporary storage. If get_segment is
/*     called on an empty list a null pointer is returned.
/*
/************************************************************************/

LINE_SEG *get_segment(n)
int n;
{
    LINE_SEG *ptr;
    int index;

    index = seg_list_head[n];
    if (index == -1)
     ptr = NULL;
    else
     {
      ptr = &seg_list[n][index];
      seg_list_head[n] = (index < 4) ? (index + 1) : 0;
```

```
 }
    return ptr;
}

/*****************************************************************/
/*
/* Procedure: get_current_segment(n)
/*
/* Description:  returns a pointer to the segment currently under
/*    construction if there is one, otherwise returns null pointer.
/*    This is accomplished by calling end_segment, copying the data
/*    into segstruct and then returning a pointer to segstruct. The
/*    memory allocated by end_segment is then freed.
/*
/*****************************************************************/

LINE_SEG *get_current_segment(n)
int n;
{
    LINE_SEG *ptr;

    if (segment_data[n].rst > 1)
      ptr = end_segment(n);
    else ptr = NULL;
    return ptr;
}

/*****************************************************************/
/*
/* Procedure: set_parameters(c1,c2,c3)
/*
/* Description:  allows the user to adjust constants which control
/*    the linear fitting algorithm. C1 is a multiplier for standard
/*    deviation and C2 is an absolute value; both are used to
/*    determine if an individual data point is usable for the
/*    algorithm. C3 is a value for ellipse thinness; it is used to
/*    determine the end of a segment. Default values are set in main.c
/*    to 3.0, 5.0, and 0.1 respectively.
/*
/*****************************************************************/

void set_parameters(c1,c2,c3)
double c1,c2,c3;
{
    C1 = c1;
    C2 = c2;
    C3 = c3;
}

/*******************************************************************/
/*
/* Procedure: enable_interrupt_operation()
/*
/* Description: places sonar control board in interrupt driven mode.
/*
/*******************************************************************/

void enable_interrupt_operation()
{
    *BIM_ptr = *BIM_ptr | 0x10;
}

/*******************************************************************/
/*
/* Procedure: disable_interrupt_operation()
/*
/* Description: stops interrupt generation by the sonar control
/*    board. A flag is set in the status register when data is ready,
/*    and it is the user's responsibility to poll the sonar system
/*    for the flag.
/*
/*******************************************************************/

void disable_interrupt_operation()
{
    *BIM_ptr = *BIM_ptr & 0xef;
}

/*******************************************************************/
/*
/* Procedure: calculate_global(n)
/*
/* Description: this procedure calculates the global x and y coordinates
/*    for the range value and robot configuration in the sonar table.
/*    The results are stored in the sonar table.
/*
/*******************************************************************/
```

```
void calculate_global(n)
int n;
{
  double lx, ly, gt, range, phi, axis, offset;

  gt = sonar_table[n].t;
  range = (float)sonar_table[n].d;
  phi = sonar_table[n].phi;
  axis = sonar_table[n].axis;
  offset = sonar_table[n].offset;

  if (range == -1) range = 9999;
  lx = sonar_table[n].x + (cos(gt + phi) * offset);     /* global x position of sonar */
  ly = sonar_table[n].y + (sin(gt + phi) * offset);     /* global y position of sonar */
  sonar_table[n].gx = lx + (cos(gt + axis) * range);   /* global x position of range */
  sonar_table[n].gy = ly + (sin(gt + axis) * range);   /* global y position of range */
 }

/******************************************************************/
/*
/* Procedure: linear_fitting(n)
/*
/* Description: this procedure controls the fitting of range data to straight
/*    line segments. First it collects three data points and establishes
/*    a line segment with it's interim data values. After the segment
/*    is established, the procedure tests each subsequent data point
/*    to determine if it falls within acceptable bounds before calling
/*    the least squares routine to include the data point in the line
/*    segment. After inclusion of the data point the segment is again
/*    tested to ensure the entire set of data points are linear enough.
/*    If any of the tests fail, the line segment is ended and a new one
/*    started. The completed line segment is stored in a data structure
/*    called segment, and segments are linked together in a linked list.
/*
/******************************************************************/

void linear_fitting(n)
int n;
{
  int sgmp, rst;
  double theta, r, sigma, delta;
  LINE_SEG *finished_segment;

  theta    = segment_data[n].theta;
```

```
r       = segment_data[n].r;
sgmp    = segment_data[n].sgmp;
rst     = segment_data[n].rst;
if (rst == 0)
 {
  segment_data[n].initx[sgmp] = sonar_table[n].gx;
  segment_data[n].inity[sgmp] = sonar_table[n].gy;
  segment_data[n].sgmp += 1;
  if (sgmp == 2)
   {
    start_segment(n);
    segment_data[n].rst = 1;
   }
 }
else
 {
  sigma = segment_data[n].sgm_delta_sq /(double) sgmp;
  delta = sonar_table[n].gx * cos(theta) + sonar_table[n].gy * sin(theta) - r;
  if ((fabs (delta) < (sigma * C1)) || (fabs (delta) < C2))
   {
   switch (rst)
{
case 1:
       segment_data[n].sgmp += 1;
       add_to_line(n, sonar_table[n].gx, sonar_table[n].gy);
       if (segment_data[n].rst == 1)
        {
         segment_data[n].rst = 2;
   }
       else if (segment_data[n].rst == 5)
       {
            reset_accumulators(n);
            segment_data[n].initx[0] = segment_data[n].initx[1];
            segment_data[n].inity[0] = segment_data[n].inity[1];
            segment_data[n].initx[1] = segment_data[n].initx[2];
            segment_data[n].inity[1] = segment_data[n].inity[2];
            segment_data[n].initx[2] = sonar_table[n].gx;
            segment_data[n].inity[2] = sonar_table[n].gy;
            segment_data[n].sgmp = 3;
            start_segment(n);
            segment_data[n].rst = 1;
 }
       break;
      case 2:
```

```
   segment_data[n].sgmp += 1;
   add_to_line(n, sonar_table[n].gx, sonar_table[n].gy);
   if (segment_data[n].rst == 5)
     {
     finished_segment = end_segment(n);
     build_list(finished_segment, n);
     reset_accumulators(n);
     segment_data[n].initx[0] = sonar_table[n].gx;
     segment_data[n].inity[0] = sonar_table[n].gy;
     segment_data[n].sgmp = 1;
     segment_data[n].rst = 0;
}
   break;
  case 3:
   segment_data[n].initx[1] = sonar_table[n].gx;
   segment_data[n].inity[1] = sonar_table[n].gy;
   segment_data[n].rst = 4;
   break;
  case 4:
   segment_data[n].sgmp += 1;
   add_to_line(n, segment_data[n].initx[1], segment_data[n].inity[1]);
   if (segment_data[n].rst == 5)
     {
     finished_segment = end_segment(n);
     build_list(finished_segment, n);
     reset_accumulators(n);
     segment_data[n].initx[2] = sonar_table[n].gx;
     segment_data[n].inity[2] = sonar_table[n].gy;
     segment_data[n].sgmp = 3;
     start_segment(n);
     segment_data[n].rst = 1;
}
   else
     {
     segment_data[n].sgmp += 1;
     add_to_line(n, sonar_table[n].gx, sonar_table[n].gy);
     if (segment_data[n].rst == 5)
       {
       finished_segment = end_segment(n);
       build_list(finished_segment, n);
       reset_accumulators(n);
       segment_data[n].initx[0] = sonar_table[n].gx;
       segment_data[n].inity[0] = sonar_table[n].gy;
       segment_data[n].sgmp = 1;
```

```
              segment_data[n].rst = 0;
 }
           else segment_data[n].rst = 2;
    }
        break;
  }
}
    else
    {
    switch (rst)
      {
  case 1:
       case 2:
         segment_data[n].initx[0] = sonar_table[n].gx;
         segment_data[n].inity[0] = sonar_table[n].gy;
         segment_data[n].rst = 3;
         break;
       case 3:
         finished_segment = end_segment(n);
         build_list(finished_segment, n);
         reset_accumulators(n);
         segment_data[n].initx[1] = sonar_table[n].gx;
         segment_data[n].inity[1] = sonar_table[n].gy;
         segment_data[n].sgmp = 2;
         segment_data[n].rst = 0;
         break;
       case 4:
         finished_segment = end_segment(n);
         build_list(finished_segment, n);
         reset_accumulators(n);
         segment_data[n].initx[2] = sonar_table[n].gx;
         segment_data[n].inity[2] = sonar_table[n].gy;
         segment_data[n].sgmp = 3;
         start_segment(n);
         segment_data[n].rst = 1;
         break;
       }
    }
  }
 }

/******************************************************************************/
/*
/* Procedure: start_segment(n)
```

```
/*
/* Description: this procedure establishes a new line segment with the three
/*    data points contained in segment_data[n].init(x and y). It writes
/*    the appropriate data to the interim values in segment_data[n].
/*
/******************************************************************************/

void start_segment(n)
int n;
{
  double theta, r, mux, muy, muxx, muyy, muxy, sds;
  int i,j;

  segment_data[n].startx = segment_data[n].initx[0];
  segment_data[n].starty = segment_data[n].inity[0];
  segment_data[n].endx   = segment_data[n].initx[2];
  segment_data[n].endy   = segment_data[n].inity[2];
  for (i = 0; i < 3; ++i)
    {
    segment_data[n].sgmx  += segment_data[n].initx[i];
    segment_data[n].sgmy  += segment_data[n].inity[i];
    segment_data[n].sgmx2 += SQR(segment_data[n].initx[i]);
    segment_data[n].sgmy2 += SQR(segment_data[n].inity[i]);
    segment_data[n].sgmxy += segment_data[n].initx[i] * segment_data[n].inity[i];
    }
  mux  = segment_data[n].sgmx/3.0;
  muy  = segment_data[n].sgmy/3.0;
  muxx = segment_data[n].sgmx2 - SQR(segment_data[n].sgmx)/3.0;
  muyy = segment_data[n].sgmy2 - SQR(segment_data[n].sgmy)/3.0;
  muxy = segment_data[n].sgmxy - (segment_data[n].sgmx * segment_data[n].sgmy)/3.0;
  theta = (atan2( -2.0 * muxy, (muyy - muxx))) / 2.0;
  r = mux * cos(theta) + muy * sin(theta);
  for (j = 0; j < 3; ++j)
   {
    sds += SQR(segment_data[n].initx[j] - mux) * SQR(cos(theta));
    sds += SQR(segment_data[n].inity[j] - muy) * SQR(sin(theta));
    sds += 2.0 * (segment_data[n].initx[j] - mux) * (segment_data[n].inity[j] - muy)
           * cos(theta) * sin(theta);
   }
  segment_data[n].sgm_delta_sq = sds;
  segment_data[n].theta = theta;
  segment_data[n].r = r;
}
```

```
/*****************************************************************************/
/*
/* Procedure: add_to_line(n, x, y)
/*
/* Description: this procedure calculates new interim data for the line segment
/*    and stores it in segment_data[n]. It also changes the end point values to
/*    the point being added.
/*
/*****************************************************************************/

void add_to_line(n,x,y)
int n;
double x, y;
{
  double sgmp;
  double m_major, m_minor, d_major, d_minor, theta, r;
  double mux, muy, muxx, muyy, muxy, sds;

  sgmp = (double)segment_data[n].sgmp;
  segment_data[n].sgmx  += x;
  segment_data[n].sgmy  += y;
  segment_data[n].sgmx2 += SQR(x);
  segment_data[n].sgmy2 += SQR(y);
  segment_data[n].sgmxy += x * y;

  mux  = segment_data[n].sgmx / sgmp;
  muy  = segment_data[n].sgmy / sgmp;
  muxx = segment_data[n].sgmx2 - SQR(segment_data[n].sgmx) / sgmp;
  muyy = segment_data[n].sgmy2 - SQR(segment_data[n].sgmy) / sgmp;
  muxy = segment_data[n].sgmxy - (segment_data[n].sgmx*segment_data[n].sgmy) / sgmp;

  m_major = (muxx + muyy)/2.0 - sqrt((muyy-muxx)*(muyy-muxx)/4.0 + SQR(muxy));
  m_minor = (muxx + muyy)/2.0 + sqrt((muyy-muxx)*(muyy-muxx)/4.0 + SQR(muxy));
  d_major = 4.0 * sqrt(fabs(m_minor/sgmp));
  d_minor = 4.0 * sqrt(fabs(m_major/sgmp));

  if ((d_minor / d_major) < C3)
   {
    theta = (atan2( -2.0 * muxy, (muyy - muxx))) / 2.0;
    r = mux * cos(theta) + muy * sin(theta);
    sds += SQR(x - mux) * SQR(cos(theta));
    sds += SQR(y - muy) * SQR(sin(theta));
    sds += 2.0 * (x - mux) * (y - muy) * cos(theta) * sin (theta);
    segment_data[n].sgm_delta_sq += sds;
    segment_data[n].theta = theta;
    segment_data[n].r = r;
    segment_data[n].endx = x;
    segment_data[n].endy = y;
    segment_data[n].d_major = d_major;
    segment_data[n].d_minor = d_minor;
   }
  else segment_data[n].rst = 5;
 }

/*****************************************************************************/
/*
/* Procedure: end_segment(n)
/*
/* Description: this procedure allocates memory for the segment data structure,
/*    loads the correct values into it and returns a pointer to the structure.
/*
/*****************************************************************************/

LINE_SEG *end_segment(n)
int n;
{
  LINE_SEG *seg_ptr;
  double startx, starty, endx, endy, delta, theta, r, length;

  seg_ptr = &segstruct;

  startx = segment_data[n].startx;
  starty = segment_data[n].starty;
  endx   = segment_data[n].endx;
  endy   = segment_data[n].endy;
  theta  = segment_data[n].theta;
  r      = segment_data[n].r;
  delta  = startx * cos(theta) + starty * sin(theta) - r;
  startx = startx - (delta * cos(theta));
  starty = starty - (delta * sin(theta));
  delta  = endx * cos(theta) + endy * sin(theta) - r;
  endx   = endx - (delta * cos(theta));
  endy   = endy - (delta * sin(theta));
  length = sqrt(SQR(startx - endx) + SQR(starty - endy));

  seg_ptr->headx  = startx;
  seg_ptr->heady  = starty;
  seg_ptr->tailx  = endx;
  seg_ptr->taily  = endy;
  seg_ptr->phi    = theta;
  seg_ptr->r      = r;
  seg_ptr->length = length;
  seg_ptr->dmajor = segment_data[n].d_major;
  seg_ptr->dminor = segment_data[n].d_minor;
  seg_ptr->sonar  = n;

  return seg_ptr;
 }

/*****************************************************************************/
/*
/* Procedure: reset_accumulators(n);
/*
/* Description: resets the accumulative values in segment_data[n] (sgmx, sgmy,
/*     sgmx2, sgmy2, sgmxy) to zero.
/*
/*****************************************************************************/

void reset_accumulators(n)
int n;
{
  segment_data[n].sgmx = 0.0;
  segment_data[n].sgmy = 0.0;
  segment_data[n].sgmx2 = 0.0;
  segment_data[n].sgmy2 = 0.0;
  segment_data[n].sgmxy = 0.0;
}

/*****************************************************************************/
/*
/* Procedure: build_list(ptr, n);
/*
/* Description: this function accepts a pointer to a segment data structure and
/*     a sonar number, and appends the segment structure to the tail of a linked
/*     list of structures for that sonar.
/*
/*****************************************************************************/

void build_list(ptr, n)
int n;
LINE_SEG *ptr;
{
  int next;

  if (seg_list_tail[n] == -1) seg_list_head[n] = 0;
  next = (seg_list_tail[n] < 4) ? ++seg_list_tail[n] : 0;
  if (next == seg_list_head[n])
    seg_list_head[n] = (seg_list_head[n] < 4) ? ++seg_list_head[n] : 0;
  seg_list[n][next] = *ptr;
  if (sonar_table[n].filetype[2] == 1)
   log_data(n,3,sonar_table[n].filenumber[2],next);
 }

/*****************************************************************************/
/*
/* Procedure: log_data(n, type, filenumber,i)
/*
/* Description: this procedure causes data to be written to a file. The filenumber
/*    designates which "column" (0,1,2, or 3) of a two dimensional array for
/*    that type of data is used. The data array and a counter for each column
/*    forms the data structure for each type. The value of i is used to index
/*    the seg_list array for storing line segments.
/*
/*****************************************************************************/

void log_data(n, filetype, filenumber,i)
int n, filetype,filenumber,i;
{
  int count,interval,next;

  switch(filetype)
   {
   case 1:
    count = raw_data_log[filenumber].count;
    interval = sonar_table[n].interval;
    if ((count < MAXRAW) && !(count % interval))
     {
      next = raw_data_log[filenumber].next;
      raw_data_log[filenumber].darray[next] = sonar_table[n].d;
      raw_data_log[filenumber].xarray[next] = sonar_table[n].x;
      raw_data_log[filenumber].yarray[next] = sonar_table[n].y;
      raw_data_log[filenumber].tarray[next] = sonar_table[n].t;
      raw_data_log[filenumber].next += 1;
     }
    raw_data_log[filenumber].count += 1;
    break;
    case 2:
     count = global_data_log[filenumber].count;
     interval = sonar_table[n].interval;
     if ((count < MAXGLOBAL) && !(count % interval))
       {
        next = global_data_log[filenumber].next;
        global_data_log[filenumber].xarray[next] = sonar_table[n].gx;
        global_data_log[filenumber].yarray[next] = sonar_table[n].gy;
        global_data_log[filenumber].next += 1;
       }
     global_data_log[filenumber].count += 1;
     break;
    case 3:
     count = segment_data_log[filenumber].count;
     if (count < MAXSEGMENT)
       {
        segment_data_log[filenumber].array[count] = seg_list[n][i];
       }
     segment_data_log[filenumber].count += 1;
     break;
    }
}


/
*********************************************************************************/
/*
/* Procedure: set_log_interval(n,d)
/*
/* Description: this procedure allows the user to set how often the sonar system
/*    writes data to the raw data or global data files. The interval d is stored
/*    at sonar_table[n], and one data point will be recorded for every d data
/*    points sensed by the sonar. Default value for interval d is 13, which for
/*    a speed of 30 cm/sec and sonar sampling time of 25 msec should record a
/*    data point every 10 cm.
/*
/
*********************************************************************************/

void set_log_interval(n,d)
int n,d;
{
  sonar_table[n].interval = d;
}
```

```
    /
***********************************************************************************/
    /*
    /* Procedure: wait_until(variable,relation,value)
    /*
    /* Description:  this procedure will delay it's completion (and thus the continuance
    /*    of the program it's embedded in) until the variable achieves the relation with
    /*    the value specified. For example, presume the robot is traveling along the x
    /*    axis. If the user wants the robot to begin redording sonar data when the x
    /*    position of the robot exceeds 500 cm., he would insert this command after the
    /*    move command:
    /*              wait_until(X,GT,500.0);
    /*              enable_sonar(sonar number);
    /*    The variable are predefined as X, Y, A and D0 through D11, and correspond to
    /*    the robot's current x position, y position, theta, and range from sonars 0
    /*    through 11. Relations are predefined as GT, LT and EQ corresponding to greater
    /*    than, less than and equal to. Value may be any numlber expressed as a double
    /*    or the predefined values PI, HPI, PI34, PI4, or DPI.
    /*
    /
***********************************************************************************/

    void wait_until(variable,relation,value)
    int variable,relation;
    double value;
    {
      double *ptr;
      double theta;
      int test,item;

      if ((variable == 14) && (relation == 17)) test = (int)(1000.0 * value);
      else if (relation == 17) test = (int)(value);

      switch (variable)
       {
       case 0:
       case 1:
       case 2:
       case 3:
       case 4:
       case 5:
       case 6:
```

```
    case 7:
    case 8:
    case 9:
    case 10:
    case 11:
      ptr = &sonar_table[variable].d;
      break;
    case 12:
      ptr = &cur_x;
      break;
    case 13:
      ptr = &cur_y;
      break;
    case 14:
      theta = 1000.0 * cur_t;
      ptr = &theta;
      break;
    }
  switch (relation)
    {
    case 15:
      do{
        item = *ptr;
        }
      while (item <= value);
      break;
    case 16:
      do{
        item = *ptr;
        }
      while (item >= value);
      break;
    case 17:
      do{
        item = (int)*ptr;
        }
      while (item != test);
      break;
    }
}

/
*******************************************************************************************
*/
```

```
    /*
    /* Procedure: xfer_raw_to_host(filenumber,filename)
    /*
    /* Description: this function allocates memory for a buffer and then converts a raw data
    /*    log file to a string format stored in the buffer. It then calls host_xfer to send
    /*    the string to the host. When that transfer is complete, it frees the memory it
    /*    allocated for the buffer. Filename must be entered in double quotes ( "dumpraw"
    /*    for example).
    /*
    /
***********************************************************************************
*/

    void xfer_raw_to_host(filenumber,filename)
    int filenumber;
    char *filename;
    {
      char *rbuffer;
      char *start;
      int i,c,j;

      i = raw_data_log[filenumber].next;
      c = 20 + (i * 33);
      rbuffer = malloc(c);
      start = rbuffer;
      for (j=0; j<i; j++)
        {
          print_flex(raw_data_log[filenumber].darray[j], rbuffer);
          print_flex(raw_data_log[filenumber].xarray[j], rbuffer);
          print_flex(raw_data_log[filenumber].yarray[j], rbuffer);
          print_flex(raw_data_log[filenumber].tarray[j], rbuffer);
          nl_flex(rbuffer);
        }
      putb('\0',rbuffer);
      rbuffer = start;
      host_xfer(rbuffer,filename);
      free(rbuffer);
     }


    /
***********************************************************************************
*/
    /*
    /* Procedure: xfer_global_to_host(filenumber,filename)
    /*
    /* Description: this function performs the same function as xfer_raw_to_host, but for
    /*     global data vice raw data.
    /*
    /
*******************************************************************************************
*/

    void xfer_global_to_host(filenumber,filename)
    int filenumber;
    char *filename;
    {
      char *gbuffer;
      char *start;
      int i,c,j;

      i = global_data_log[filenumber].next;
      c = 20 + (i * 17);
      gbuffer = malloc(c);
      start = gbuffer;
      for (j=0; j<i; j++)
        {
          print_flex(global_data_log[filenumber].xarray[j], gbuffer);
          print_flex(global_data_log[filenumber].yarray[j], gbuffer);
          nl_flex(gbuffer);
        }
      putb('\0', gbuffer);
      gbuffer = start;
      host_xfer(gbuffer,filename);
      free(gbuffer);
    }

    /
*******************************************************************************************
*/
    /*
    /* Procedure: xfer_segment_to_host(filenumber,filename)
    /*
    /* Description: this function performs the same function as xfer_raw_to_host, but for
    /*     segment data vice raw data.
    /*
```

```
/
***************************************************************************************
*/

void xfer_segment_to_host(filenumber,filename)
int filenumber;
char *filename;
{
  char *segbuffer;
  char *start;
  int i,c,j;

  i = segment_data_log[filenumber].count;
  c = 20 + (i * 77);
  segbuffer = malloc(c);
  start = segbuffer;
  for (j=0; j<i; j++)
    {
      print_flex(segment_data_log[filenumber].array[j].headx, segbuffer);
      print_flex(segment_data_log[filenumber].array[j].heady, segbuffer);
      print_flex(segment_data_log[filenumber].array[j].tailx, segbuffer);
      print_flex(segment_data_log[filenumber].array[j].taily, segbuffer);
      nl_flex(segbuffer);
      print_flex(segment_data_log[filenumber].array[j].phi, segbuffer);
      print_flex(segment_data_log[filenumber].array[j].r, segbuffer);
      print_flex(segment_data_log[filenumber].array[j].length, segbuffer);
      print_flex(segment_data_log[filenumber].array[j].dmajor, segbuffer);
      print_flex(segment_data_log[filenumber].array[j].dminor, segbuffer);
      nl_flex(segbuffer);
    }
  putb('\0',segbuffer);
  segbuffer = start;
  host_xfer(segbuffer,filename);
  free(segbuffer);
 }


/
***************************************************************************************
*/
/*
/* Procedure: host_xfer(buffer,filename)
/*
/* Description: this function transfers a data string from the buffer to the host. Not a
```

```
    /*  user function; is called by data conversion functions such as xfer_raw_to_host.
    /*  User would call the xfer_raw_to_host (or equivalent for global or segment data)
    /*  to download data from the robot.
    /*
    /
*****************************************************************************************
*/

    void host_xfer(buffer,filename)
    char *buffer;
    char *filename;
    {
      i_port(HOST, 9600, 0, 0, 0);
      r_printf("\12\15 connect cable and keyin\" \"");
      while(r_getchar() != ' ');
      putstr("\n",HOST);
      i_port(HOST, 9600, 0, 0, 1);
      r_printf("\12\15 ready for dump ");
      while(r_getchar() != 'g');
      putstr("ytof ",HOST);
      putstr(filename,HOST);
      putstr(" w \n",HOST);
      while(r_getchar() != ' ');
      r_printf("dumping ");
      putstr(buffer,HOST);
      putb('\4',HOST);
      putb('\4',HOST);
      r_printf("\7\7\7");
      return;
     }


    /
**************************************************************************************/
    /*
    /* Procedure: finish_segments(n)
    /*
    /* Description: this function completes segments at the end of a data run. Necessary
    /*   because the linear fitting function only terminates a segment based on the
    /*   data - it has no way of knowing that the user has stopped collecting data.
    /*
    /
**************************************************************************************/
```

```
void finish_segments(n)
int n;
{
  LINE_SEG *finished_segment;

  finished_segment = end_segment(n);
  build_list(finished_segment, n);
}
```

APPENDIX B - INTERRUPT HANDLER CODE

```
###########################################################################
#
# Procedure: ih_sonar
#
# Description: Interrupt handler for the sonar control board. Loads
# group number, contents of the four data registers, the overflow
# word, current theta, current y and current x onto the stack in
# that order. It then calls the serve_sonar function in sonar.c
# which places the data in the appropriate data structures. Overflow
# word is simply a four bit concatenation of the overflow bits in
# the individual data registers. Data register contents are
# masked to allow only the actual range data.
#
#
###########################################################################

        .globl      _ih_sonar

status  = 0xffff83f9
data1   = 0xffff83f0
stop    = 0xffff83f8
        .text
_ih_sonar:
        moveml      d0-d7/a0-a5, sp@-           |save register contents before use
        link        a6, #-184                   |stack space for coprocessor status
        fsave       a6@(-184)                   |save coprocessor status
        fmovemx     fp0-fp7, _fpx_save_sonar    |save fp registers
        fmovel      fpcr, _fpcr_save_sonar      |save fp control register
        fmovel      fpsr, _fpsr_save_sonar      |save fp status register
        fmovel      fpiar, _fpia_save_sonar     |save fp iaddr register
        clrl d0
        movl        #status, a0                 |load address of status register
        movb        a0@, d0                     |load status register into d0
        andl        #0x00000018, d0             |extract group # from status
        lsrl        #3, d0                      |shift group number over to the right
        movl        d0, sp@-                    |push group # onto stack
        clrl        d2
        movl        #data1, a0                  |load address of data register #1
xferdata:
        movw        a0@, d1                     |move register contents into d1
        movl     d1, sp@-                    |push d1 onto stack
        andl     #0x00000fff, sp@            |mask out all but last twelve bits
        lsll     #1, d2                      |shift overflow word left 1 bit
        andl     #0x00008000, d1             |extract bit 15 of register (ovfl)
        tstl     d1                          |test for overflow
        beq      gooddata                    |if no overflow branch
        orl      #1, d2                      |sets overflow word lsb if overflow
gooddata:
        addql    #2, a0                      |increment address for next register
        cmpl     #stop, a0                   |stop when address is ffff83f8
        bne      xferdata                    |continue data transfer
        movl     d2, sp@-                    |push overflow word onto stack
        movl     _cur_t+4, sp@-              |push theta onto stack. Two pushes
        movl     _cur_t, sp@-                | to xfer 64 bit data
        movl     _cur_y+4, sp@-              |push current y onto stack
        movl     _cur_y, sp@-
        movl     _cur_x+4, sp@-              |push current x onto stack
        movl     _cur_x, sp@-

        jsr      _serve_sonar                |link to C routines

        addl     #48, sp                     |remove parameters from stack
        fmovel   _fpia_save_sonar, fpiar     |restore fp iaddr register
        fmovel   _fpsr_save_sonar, fpsr      |restore fp status register
        fmovel   _fpcr_save_sonar, fpcr      |restore fp control register
        fmovemx  _fpx_save_sonar, fp0-fp7    |restore fp registers
        frestore a6@(-184)                   |restore coprocessor status
        unlk     a6                          |clear up stack
        moveml   sp@+, d0-d7/a0-a5           |restore registers
        rte
```

APPENDIX C - TEST PROGRAMS

1. Operating Individual Sonars

```
/*********************************************************
 read individual sonars
   August 10, 1991 by Sherfey
*********************************************************/
#include "mml.h"

user()
{
   int n,i,d;

   i=1;

   r_printf("\12\15 input sonar no.");
   n=getint(CONSOLE);
   enable_sonar(n);
   do
   {
       for (i=0; i<20; i++) wait_sonar(n); /*produces a range every half second*/
        d = (int)sonar_table[n].d;
        if (d < 0) r_printf("\12\15 Overranged");
        else
            {
            r_printf("\12\15");
            r_printfi(d);
            }
   }
   while(1==1);

}
```

2. Manipulating Multiple Sonars

```
/*********************************************************/
 manipulate sonars
   August 12, 1991 by Sherfey
/*********************************************************/
#include "mml.h"

user()
{
   int n,m,i,j,k,l,s,d,x;

    i=1;
    j=1;
    k=20;

    do
    {
        switch (j)
        {
        case 1:
             r_printf("\12\15 input number of sonar to enable: ");
             n=getint(CONSOLE);
            enable_sonar(n);
             break;
         case 2:
            r_printf("\12\15 input number of sonar to disable: ");
             m=getint(CONSOLE);
             disable_sonar(m);
             break;
         case 3:
             r_printf("\12\15 input number of wait cycles: ");
             k=getint(CONSOLE);
             break;
         }
        do
         {
            for (s=0; s<16; s++)
             {
                    if (enabled_sonars[s] == 1)
                    {
                         for (i=0; i<k; i++) wait_sonar(s);
                         d = (int)sonar_table[s].d;
                         r_printf(" ");
                         r_printfi(s);
                         r_printf("/");
                         r_printfi(d);
                    }
               }
               r_printf("\12\15");
              x = (r_getchar() != ' ') ? 1 : 0;
          }
          while(x);
          r_printf("\12\15\15 0 to quit, 1 to enable, 2 to disable, 3 to change wait : ");
         j=getint(CONSOLE);
     }
     while(j != 0);

}
```

3. Test Runs In Corridor

```
/*****************************************************************/
/* Gather global and segment data on straight run of 6 meters
/* by Sol Sherfey August 29, 1991
/*****************************************************************/
#include "mml.h"

user()
{
 POSTURE p1, p2, p3;

 def_posture(0.0, 0.0, HPI, &p1);
 def_posture(0.0, 600.0, HPI, &p2);
 def_posture(0.0, 650.0, HPI, &p3);

 set_parameters(2.0, 2.0, 0.08);
set_log_interval(7,5);
set_rob(&p1);

enable_sonar(7);
mark_motion();
move(&p2);
move(&p3);
enable_linear_fitting(7);
enable_data_logging(7,1,0);
enable_data_logging(7,2,0);
wait_motion();
disable_sonar(7);
finish_segments(7);
disable_linear_fitting(7);
disable_data_logging(7,2);
disable_data_logging(7,1);
motor_on = NO;
xfer_global_to_host(0,"test_global8");
xfer_segment_to_host(0,"test_segment8");
}
```

4. Dynamic Test On Blocks (Non-Moving)

```
/*********************************************************************/
/* generate and record segments while rolling straight
/* for 1000 cm and moving the barrier
/* by Sol Sherfey 14 August, 1991
/*********************************************************************/
#include "mml.h"

extern LINE_SEG *end_segment();

user()
{
   POSTURE p, p1, p2;
   int i,c,z,n;
   LINE_SEG *finished_segment;

   reset_accumulators(7);
```

```
set_rob(def_posture(0.0, 0.0, 0.0, &p)); /* Initial posture */
enable_sonar(7);
move(def_posture(1000.0, 0.0, 0.0, &p1));
enable_linear_fitting(7);
enable_data_logging(7,2,0);
stop(def_posture(1050.0, 0.0, 0.0, &p2));
do
    {
        z = (int)sonar_table[7].x;
    }
while (z < 1000);
disable_sonar(7);
finished_segment = end_segment(7);
build_list(finished_segment,7);
disable_data_logging(7,2);
r_printf("\12\15\15");
c = segment_data_log[0].count;
r_printf("count = ");
r_printfi(c);
r_printf(" x posit = ");
r_printfi((int)sonar_table[7].x);
r_printf(" y posit = ");
r_printfi((int)sonar_table[7].y);
r_printf(" range = ");
r_printfi((int)sonar_table[7].d);
r_printf("\12\15\15");

for (i=0; i<c; i++)
{
    r_printf(" hx = ");
    r_printfi((int)segment_data_log[0].array[i].headx);
    r_printf(" hy = ");
    r_printfi((int)segment_data_log[0].array[i].heady);
    r_printf(" tx = ");
    r_printfi((int)segment_data_log[0].array[i].tailx);
    r_printf(" ty = ");
    r_printfi((int)segment_data_log[0].array[i].taily);
    r_printf(" length = ");
```

```
        r_printfi((int)segment_data_log[0].array[i].length);
        r_printf("\12\15");
    }
}
```

5. Another Dynamic Test Program

```
/*******************************************************************/
/* generate and record segments while rolling straight
/* test the wait_until function
/* by Sol Sherfey 14 August, 1991
/*******************************************************************/
#include "mml.h"

extern LINE_SEG *end_segment();

user()
{
    POSTURE p, p1, p2,p3,p4;
    int i,c,z,n;
    LINE_SEG *finished_segment;

   reset_accumulators(7);
   set_rob(def_posture(0.0, 0.0, 0.0, &p)); /* Initial posture */
   enable_sonar(7);
   move(def_posture(300.0, 0.0, 0.0, &p1));
   enable_linear_fitting(7);
   enable_data_logging(7,2,0);
   wait_until(X,GT,300.0);
   disable_data_logging(7,2);
   disable_linear_fitting(7);
   move(def_posture(400.0,100.0,HPI,&p2));
   move(def_posture(400.0,400.0,HPI,&p3));
   enable_linear_fitting(7);
   enable_data_logging(7,2,0);
   stop(def_posture(400.0, 450.0, HPI, &p4));
   wait_until(Y,GT,400.0);
   disable_sonar(7);
   finished_segment = end_segment(7);
```

```
    build_list(finished_segment,7);
    disable_data_logging(7,2);
    r_printf("\12\15\15");
    c = segment_data_log[0].count;
    r_printf("count = ");
    r_printfi(c);
    r_printf(" x posit = ");
    r_printfi((int)sonar_table[7].x);
    r_printf(" y posit = ");
    r_printfi((int)sonar_table[7].y);
    r_printf(" range = ");
    r_printfi((int)sonar_table[7].d);
    r_printf("\12\15\15");
    for (i=0; i<c; i++)
    {
        r_printf(" hx = ");
        r_printfi((int)segment_data_log[0].array[i].headx);
        r_printf(" hy = ");
        r_printfi((int)segment_data_log[0].array[i].heady);
        r_printf(" tx = ");
        r_printfi((int)segment_data_log[0].array[i].tailx);
        r_printf(" ty = ");
        r_printfi((int)segment_data_log[0].array[i].taily);
        r_printf(" length = ");
        r_printfi((int)segment_data_log[0].array[i].length);
        r_printf("\12\15");
    }
}
```

REFERENCES

1 Burks, B. L., de Saussure, G., Weisbin, C. R., Jones, J. P., Hamel, W. R., "Autonomous Navigation, Exploration, and Recognition Using the HERMIES-IIB Robot", *IEEE Expert*, Winter 1987.

2 Crowley, J. L., "Dynamic World Modeling for an Intelligent Mobile Robot Using a Rotating Ultra-Sonic Ranging Device", *Proceedings of the IEEE International Conference on Robotics and Automation*, St. Louis, Missouri, March 1985, pp. 128-135.

3 Crowley, J. L., "World Modeling and Position Estimation for a Mobile Robot Using Ultrasonic Ranging", *Proceedings of the IEEE International Conference on Robotics and Automation*, Scottsdale, Arizona, May 1989, pp. 674-680.

4 Drumheller, M., "Mobile Robot Localization Using Sonar", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-9, no. 2, pp. 325-332, March 1987.

5 Elfes, A., "Sonar-Based Real-World Mapping and Navigation", *IEEE Journal of Robotics and Automation*, vol. RA-3, no. 3, pp. 149-165, 1987.

6 Floyd, C., Kanayama, Y., Magrino, C., "Underwater Obstacle Recognition Using a Low-Resolution Sonar", *Proceedings of the Seventh International Symposium on Unmanned Untethered Submersible Technology*, September 1991.

7 Hartman, B., Kanayama, Y., Smith, T., "Model and Sensor Based Precise Navigation by an Automous Mobile Robot", *Proceedings of the International Conference on Advanced Robotics*, Columbus, Ohio, March 1989, pp. 98-109.

8 Kanayama, Y., Noguchi, T., "Spatial Learning by an Automous Mobile Robot with Ultrasonic Sensors", Univ. of California Santa Barbara Dept. of Comp. Sci. *Technical Report TRCS89-06*, February 1989.

9 Kanayama, Y., Noguchi, T., "Locomotion Functions for a Mobile Robot Language", *Proc. IEEE/RSJ International Workshop on Intelligent Robots and Systems*, pp. 542-549, Tsukuba, Japan, September 4-6, 1989.

10 Kanayama, Y., Onishi, M., "Locomotion Functions in the Mobile Robot Language, MML", *Technical Report of Naval Postgraduate School*, 1990.

11 Kuc, R., Siegel, M. W., "Physically Based Simulation Model for Acoustic Sensor Robot Navigation", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-9, no. 6, pp.766-778, November 1987.

12 Kuc, R., "A Spatial Sampling Criterion for Sonar Obstacle Detection", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 7, pp.686-690, July 1990.

13 Kwak, S. H., Ong, S. M., McGhee, R. B., "A Mission Planning Expert System for an Autonomous Underwater Vehicle", *Proceedings of AUV 90 Conference*, Washington D.C., June 5-6, 1990.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943	2
Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
Dr. Yutaka Kanayama, Code CS/Ka Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
Dr. Man-Tak Shing, Code CS/Sh Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
LT. Solomon R. Sherfey 1135 Fifth St. Apt. 1 Monterey, CA 93940	2